UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

SDK para la comunicación de dispositivos móviles con módulos Digi ConnectCore® mediante Bluetooth Low Energy

Autor/es

ANDRÉS SÁENZ GALÁN

Director/es

María Vico Pascual Martínez Losa

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2019-20

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

SDK para la comunicación de dispositivos móviles con módulos Digi ConnectCore® mediante Bluetooth Low Energy

Realizado por:

Andrés Sáenz Galán

Tutelado por:

María Vico Pascual Martínez-Losa

Logroño, junio, 2020

Resumen

Digi International desarrolla pequeñas computadoras compactas ConnectCore como parte de sus soluciones destinadas a aplicaciones IoT. Dichas computadoras ofrecen una amplia variedad de interfaces para la comunicación tanto cableada como inalámbrica. Concretamente, la interfaz Bluetooth Low Energy (BLE) le permite transmitir datos a otros dispositivos de forma inalámbrica manteniendo un consumo de energía muy bajo.

Sin embargo, no todos los modelos de ConnectCore dan soporte a esta tecnología, pues es un aspecto que encarece el producto final. Por otra parte, estos dispositivos poseen un socket para conectarse a un pequeño módulo de radiofrecuencia de Digi llamado XBee que sí soporta esta tecnología, por lo que la carencia del ConnectCore podría suplirse si fuera posible hacer uso de la interfaz BLE del XBee conectado.

El objetivo de este proyecto consiste en crear un SDK que ofrezca a clientes de Digi la posibilidad de utilizar fácilmente BLE para el intercambio de datos entre un ConnectCore y otro dispositivo BLE cliente, ya sea a través de la interfaz nativa de la computadora o de la interfaz de un XBee 3 conectado a ella, y abstrayéndolo de tal forma que el usuario final no tenga que preocuparse por qué interfaz sea la utilizada.

Este SDK estará compuesto de un API en Python que posea los métodos necesarios para llevar esa comunicación BLE de forma totalmente segura, y de un ejemplo de uso sencillo formado por dos aplicaciones, una aplicación Python embebida en el ConnectCore y una aplicación móvil multiplataforma para Android e iOS.

Abstract

Digi International develops small, compact computers, ConnectCores, as part of their solutions for IoT applications. These computers offer a wide variety of interfaces for wired and wireless communication. Specifically, the Bluetooth Low Energy (BLE) interface allows for wireless data transfer to other devices, while providing considerably reduced power consumption.

However, not all ConnectCore models support this technology, since it is an aspect that raises the price of the final product. On the other hand, these devices have a socket that connects to a small radio frequency module called XBee, also developed by Digi, that does support it, and so that shortage could be overcome if it were possible to use the BLE interface of the connected XBee.

The objective of this project is to create an SDK that offers Digi clients the possibility to easily use BLE for data exchange between a ConnectCore and another BLE client device, either through the computer's native interface or through the interface of the connected XBee, and in such a way that the end user does not have to worry about which one is being used.

The SDK will be composed of a Python API that contains the methods required to carry that BLE communication in a totally secure way, and a simple example of use formed by two applications, an embedded Python application for the ConnectCore, and a multiplatform app for Android and iOS.

Tabla de contenidos

Capítulo 1. Introducción

Digi International es una empresa multinacional, pionera en el mundo de la comunicación inalámbrica, especializada en ofrecer soluciones IoT y M2M en una amplia diversidad de áreas tales como la industria energética o las Smart Cities. Con este objetivo, Digi se encarga del desarrollo tanto hardware como software de una serie de productos que permiten a sus clientes crear entornos IoT seguros y bien monitorizados, entre los que se incluyen desde routers celulares hasta sistemas embebidos.

Uno de los protocolos que utilizan como interfaz de comunicación entre dispositivos IoT es Bluetooth Low Energy (BLE)[1]. Este protocolo se caracteriza por garantizar un bajo consumo de energía y un coste reducido con todas las ventajas del Bluetooth tradicional.

El presente proyecto nace del deseo de conseguir que dicha comunicación a través de BLE sea más fácil de implementar por los usuarios de productos Digi, lo que permita además comunicarse de forma sencilla con dispositivos móviles, que suponen una interfaz Bluetooth muy común y a la que se puede acceder en cualquier momento y lugar.

En la siguiente memoria se expone el proceso seguido en la elaboración del proyecto "SDK para la comunicación de dispositivos móviles con módulos Digi ConnectCore mediante Bluetooth Low Energy", incluyendo aspectos tales como la planificación y el seguimiento de las actividades realizadas.

1.1 Contexto

Como parte de sus soluciones industriales IoT, Digi ofrece varias gamas de módulos para sistemas embebidos. Dos de estas son especialmente relevantes para el proyecto: la gama ConnectCore y la gama Digi XBee.

Los ConnectCore (Figura 2) son pequeñas computadoras compactas para aplicaciones industriales IoT. Ofrecen múltiples interfaces de comunicación inalámbrica, entre las que se encuentra BLE. Además, son compatibles con una diversidad de distribuciones de Linux.

Por otra parte, los XBee (Figura 1) son pequeños módulos para la comunicación por radio frecuencia que permiten crear redes para la transmisión de datos en conexiones punto a punto,

[1] Bluetooth Low Energy (BLE) es un subconjunto del Bluetooth clásico que se introdujo en la especificación del núcleo de Bluetooth en su versión 4.0, y que cuenta con una pila de protocolos completamente diferente orientada a conexiones sencillas en aplicaciones de muy baja potencia.

Dentro de esta pila, interesan especialmente dos capas denominadas GAP y GATT.

La capa GAP (Generic Access Profile) controla cómo un dispositivo se anuncia y de qué forma puede interactuar con otros dispositivos. Define varios roles diferentes: el Periférico, de tamaño pequeño y bajo consumo; y el Central, con mayor potencia y memoria.

La capa GATT (Generic Attribute Profile) define cómo dos dispositivos BLE transfieren información entre sí, una vez establecida la comunicación. Durante una transacción GATT, el Periférico se denomina servidor GATT, y el Central se denomina cliente GATT. La forma en que los datos se almacenan en un servidor GATT sigue una estructura concreta: una serie de datos encapsulados cada uno en una Característica, que tiene propiedades variables y un UUID único, y una serie de Servicios que contienen una o varias de esas Características, también con UUID único.

punto a multipunto y redes en malla, todo ello con bajo coste y consumo. En su última versión, los XBee 3, utilizan estándares de comunicación tales como Zigbee, DigiMesh o IEEE 802.15.4, e introducen como novedad la posibilidad de cambiar entre esos protocolos sin necesidad de modificar el hardware, además de la nueva inclusión de Bluetooth Low Energy y de la posibilidad de programarlos directamente por medio de micropython.

Además, es importante tener en cuenta que estos dos dispositivos pueden conectarse entre sí haciendo uso del socket XBee presente en los ConnectCore para compartir sus funcionalidades.

El principal problema que motiva este proyecto viene del hecho de que existen variantes de los dispositivos ConnectCore que no tienen ningún tipo de soporte Bluetooth. Esto es así porque dicho soporte encarece el producto final, y puede no ser relevante en algunos casos. Pero los ConnectCore tienen la capacidad de conectarse a un XBee 3 que sí soporta el protocolo. Si fuera posible hacer uso del XBee para suplir la carencia del ConnectCore, se ahorraría al cliente la molestia de tener que adquirir un módulo diferente aun cuando ya posee todas las piezas necesarias para la comunicación BLE.

Pero la forma de acceder a las funciones BLE de un ConnectCore con soporte y de uno sin soporte conectado a un XBee son muy diferentes, y un XBee ya ofrece un servicio BLE predeterminado por defecto, algo que no sucede con el otro módulo. Para poder llegar al escenario deseado, la solución consistiría en abstraer toda la comunicación BLE, igualando el acceso a la interfaz en los dos entornos.

Figura 1. Digi XBee 3 THT

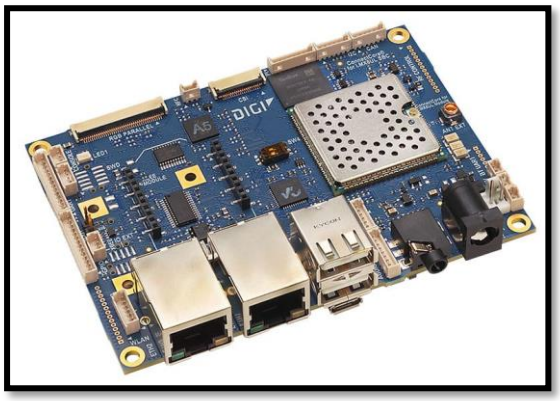

Figura 2. Digi ConnectCore 6UL SBC Pro, con un socket para XBee

1.2 Antecedentes

La idea del proyecto fue propuesta por Digi a finales de octubre, durante el periodo de realización de las prácticas externas que tuvo lugar entre el mes de septiembre y de diciembre de 2019.

En estos meses tuvo lugar la formación en varias de las tecnologías clave para el proyecto:

- Se estudió el funcionamiento de BLE, así como la estructura de los servidores GATT, y se desplegó uno en un ConnectCore 6UL compatible con Bluetooth.

- Se definió un prototipo de protocolo de comunicación a través de la interfaz BLE que tuviera en cuenta aspectos de seguridad combinando el protocolo SRP[2] con AES en modo CTR[3], tal como ocurre en la comunicación con los XBee.

- Se desarrolló una aplicación móvil básica capaz de actuar como cliente GATT para comunicarse con cualquiera de los productos.

En la mayoría de tareas realizadas, se hizo uso de las librerías propias que Digi utiliza para trabajar con sus dispositivos, como las librerías Python y C# de XBee, y en casos en que no soportaran algunas de las funcionalidades deseadas se accedió a librerías de terceros. Esto se verá más detenidamente en la sección de Tecnologías, posteriormente.

Cabe destacar que actualmente ni las librerías utilizadas en este periodo ni ninguna otra librería interna permiten abstraer el uso de BLE. Preparar la infraestructura BLE hasta ahora requería hacer uso de la librería Bluezero en Python. Al finalizar el proyecto desaparecerá la necesidad que tiene actualmente el usuario final de acceder a librerías Python externas y de estudiar su funcionamiento.

1.3 Objetivos

En la sección 1.1 se han visto los principales aspectos que dan origen al proyecto, que giran en torno a la idea de aumentar la comodidad y extender las capacidades de BLE en los ConnectCore de Digi. Para tratar con todo lo mencionado, el propósito principal será el de desarrollar un SDK que consiga la fácil comunicación entre dispositivos móviles y un módulo ConnectCore a través de la interfaz Bluetooth Low Energy.

El SDK contará con tres componentes principales, cada una con unos objetivos específicos:

En primer lugar, un API en Python que:

- Abstraiga la interfaz BLE usada por los módulos ConnectCore de forma nativa, o que utilizan la interfaz de un XBee, y los trate de igual forma desde el punto de vista del usuario. Como un XBee ya tiene un servidor GATT predefinido, debe poder crear uno igual en un ConnectCore.

- Facilite el uso de la interfaz BLE y que maneje todo aquello relacionado con el anunciamiento del servicio.

---

[2] SRP (secure remote password) se trata de un protocolo Zero-Knowledge de autenticación y distribución de claves utilizado en modelos cliente-servidor.

Esto quiere decir que el protocolo SRP asegura que un atacante situado entre las dos partes no pueda obtener suficiente información para adivinar la contraseña, y asegura además que el servidor no almacena ninguna información relacionada con la contraseña que pudiera ser robada.

[3] Advanced Encryption Standard o AES, es un algoritmo simétrico de cifrado por bloques y uno de los más utilizados en criptografía de clave simétrica. Según la longitud de la clave en bits, se habla de AES-128, AES-192 o AES-256. El modo CTR se refiere al modo de operación utilizado, que describe cómo aplicar repetidamente la operación de cifrado a los sucesivos bloques para transformar de forma más segura cantidades de datos superiores a un bloque. En concreto, CTR utiliza un contador que se va actualizando en sucesivos cifrados de bloque, partiendo de un valor inicial (nonce) aleatorio.

- Permita la comunicación entre cliente y servidor GATT a través de un protocolo que defina el propio API.

- Añada una capa extra de seguridad sobre la interfaz BLE.

En segundo lugar, una aplicación Python que utilice el API anterior para recoger y enviar datos entre un fichero de propiedades del servidor y un cliente a través de la interfaz BLE. Su objetivo será el de ofrecer un caso de uso sencillo del API en un servidor GATT a modo de ejemplo.

Por último, una aplicación móvil multiplataforma para Android e iOS que se comunique a través de la interfaz BLE con módulos ConnectCore y que sea capaz de modificar y obtener los contenidos del fichero de propiedades anterior. Esta aplicación servirá para mostrar el uso del API desde el lado del cliente.

Además de todo esto, y si hubiera tiempo suficiente, se incluirá como objetivo extra el desarrollo de una librería Python compuesta por todas aquellas clases y métodos de la librería de XBee que hubieran sido utilizadas en el desarrollo general del proyecto, con el fin de facilitar futuras tareas similares.

Para que el producto final pueda ser aceptado por el cliente, debe cumplir una serie de requisitos mínimos, los cuales se listan a continuación. Además de estos, se consideran también aquellos que, aunque no sean imprescindibles, añaden valor extra al resultado.

Requisitos mínimos del API Python:

- Debe poder crear un servidor GATT igual al de un XBee.
- Debe permitir a un servidor GATT anunciar sus servicios.
- Debe permitir a un servidor GATT emparejarse con un cliente.
- Debe permitir transmitir datos entre cliente y servidor GATT.
- Para transmitir datos debe encapsularlos con un protocolo propio.
- Debe garantizar que la comunicación a través de BLE sea transparente para el usuario.
- Debe garantizar que el usuario pueda utilizar el API de igual forma indistintamente del medio físico (ConnectCore con soporte BLE nativo o ConnectCore conectado a XBee).
- Debe asegurar que la comunicación a través de la interfaz BLE sea segura.

Requisitos mínimos de la aplicación Pyhton de ejemplo:

- Debe utilizar el API para anunciar sus servicios.
- Debe utilizar el API para emparejarse a un cliente.
- Debe utilizar el API para obtener y transmitir datos entre un fichero de propiedades y el cliente a través de la interfaz BLE.
- Debe poder obtener datos desde un fichero de propiedades.
- Debe poder utilizar los datos obtenidos para modificar un fichero de propiedades.

Requisitos mínimos de la aplicación móvil multiplataforma:

- Debe realizar un escaneo de dispositivos BLE cercanos.
- Debe listar los dispositivos descubiertos.
- Debe permitir autenticarse y emparejarse a un dispositivo ConnectCore.

- Debe permitir enviar y recibir datos correspondientes al fichero de propiedades del servidor BLE.
- Debe mostrar los datos recibidos y enviar por medio de una interfaz gráfica.

Requisitos adicionales:

- Incluir una librería adicional que recoja o encapsule métodos de la librería XBee que sean más relevantes para el proyecto.
- Extender las funcionalidades de la aplicación móvil.
- Mejorar la interfaz gráfica de la aplicación móvil.

1.4 Metodología

Para la gestión y el desarrollo del producto, se va a utilizar una metodología ágil basada en Scrum, pero con ciertas modificaciones en sus principios generales para ajustarlo a las características excepcionales del proyecto (su naturaleza académica, un equipo reducido, etc.). Los roles principales se mantienen, y serán asignados de la siguiente forma:

- **Product Owner**: Pedro Pérez, responsable de Digi International Spain.
- **Scrum Master**: Diego Escalona, Héctor González y Rubén Moral, desarrolladores en Digi International Spain y tutores del estudiante.
- **Equipo de trabajo**: Andrés Sáenz.

Respecto a la gestión del tiempo, cada sprint tendrá una duración de 40 horas, repartidas en dos semanas. Esto significa trabajar 4 horas al día, 5 días a la semana.

Uno de los cambios respecto a un Scrum más tradicional viene con las reuniones. Al comienzo de cada sprint, habrá una reunión de Sprint Planning para planificar las tareas correspondientes. Como equivalente al Scrum Daily Meeting, se realizarán pequeñas reuniones tres días a la semana. Finalmente, al final de cada sprint se llevará a cabo una reunión Sprint Review en la que se muestre el estado del producto. Esta última se mezclará con las funciones de una reunión Sprint Restrospective, para detectar qué tal ha ido el trabajo durante el Sprint.

1.5 Tecnologías

A lo largo del desarrollo del proyecto, se esperan utilizar varias tecnologías diferentes. Algunas de ellas son requeridas debido a factores tales como la metodología seguida en la empresa, pero otras ofrecieron más libertad en su elección, por lo que fue necesario valorar las distintas alternativas.

En primer lugar, los principales lenguajes utilizados serán los siguientes:

- **Python**: para el API y la aplicación de ejemplo, dado que su uso principal estará en los ConnectCore que soportan este lenguaje. El IDE para trabajar con Python será PyCharm.

- **C#**: para la aplicación móvil, pues es el lenguaje principal de Xamarin. Además de este se utilizará el lenguaje de marcado **XAML** para la interfaz gráfica. Para trabajar con C# se utilizará Visual Studio 2019.

Otras tecnologías que también serán necesarias son:

- **Bluezero**: es una librería de Python que permite un acceso sencillo a Bluez, la pila Bluetooth oficial de Linux. Este es el API utilizada actualmente en la empresa en todos los desarrollos relacionados con Bluetooth en un entorno Linux, de modo que se usa en este proyecto para mantener la consistencia.

- **Xamarin.Forms**: el entorno para el desarrollo de la aplicación móvil. Debido a la amplia selección de entornos de desarrollo multiplataforma disponibles, durante el periodo de prácticas se llevó a cabo un análisis de alternativas para determinar cuál sería el más adecuado. Los entornos seleccionados para esta comparativa fueron Apache Cordova, Flutter, Gluon Mobile, React Native y Xamarin.Forms, algunos por petición directa de los tutores y otros por tener especial relevancia en la escena actual.

  Los primeros en ser descartados fueron Apache Cordova y Gluon, el primero por estar centrado en el desarrollo de aplicaciones híbridas que reducen considerablemente el rendimiento y complican el testeo, y el segundo por tener una comunidad demasiado pequeña y utilizar JavaFX, una tecnología muy poco extendida en desarrollo móvil.

  React Native y Flutter demostraron ser las mejores opciones respecto a que sus aplicaciones son casi nativas y a que ofrecen la mejor experiencia a los desarrolladores. Sin embargo, el uso de las funciones de Bluetooth en estos entornos resultó complicado por requerir código nativo en React Native y por no tener librerías fiables en Flutter.

  Xamarin.Forms, pese a ofrecer un rendimiento ligeramente inferior al de las anteriores, compensa por contar con una lista mayor de plugins fáciles de utilizar, entre los que se encuentra Plugin.BLE, que da soporte a Bluetooth Low Energy. Además, ofrece la posibilidad de trabajar con iOS de forma eficiente desde Windows mediante el emparejamiento con Mac. Por último, al trabajar con C#, Xamarin es compatible con las librerías XBee para C# de Digi.

- **Plugin.BLE**: el plugin necesario para acceder a la funcionalidad Bluetooth en un entorno Xamarin.

- **Mac y iPhone**: para el desarrollo de la parte de iOS de la aplicación multiplataforma en Xamarin.Forms es necesario disponer tanto de un iPhone en el que desplegar la aplicación como de un Mac que esté en la misma red local para compilar la aplicación.

- **Git**: será el software de control de versiones utilizado. El repositorio estará alojado en un servidor de Stash (ahora Bitbucket). Se utilizará a través de línea de comandos y mediante la interfaz gráfica Git Extensions.

- **JIRA**: es una herramienta especializada en la gestión de proyectos ágiles. Se utilizará por estar ya integrada en la empresa y por ser idónea para un proyecto Scrum como este.

- **Los módulos ConnectCore y XBee**, ya explicados en la sección 1.1.

- **Tera Term:** un software emulador de terminal para la comunicación con los módulos ConnectCore. Su elección viene del hecho de que ya fue utilizado durante el periodo de prácticas, por lo que no será necesario un aprendizaje extra.

- **Un portátil con Kubuntu como sistema operativo y Bluetooth integrado**, para facilitar el desarrollo y el testeo del API Python y la aplicación de ejemplo.

1.6 Planificación inicial

En la siguiente sección se concreta el plan a seguir durante el plazo de desarrollo del proyecto. Este plan abarca la descomposición en tareas de los principales objetivos y la asignación de tiempos estimados a cada una de estas tareas.

La estancia en la empresa tiene lugar entre el 3 de febrero y el 29 de mayo, así que la estructura del proyecto será la siguiente: las 3 semanas iniciales son para la introducción de la memoria y la planificación; el resto de semanas hasta el 25 de mayo se dividen para formar 6 sprints de 2 semanas cada uno (con excepción del sprint que abarque los días festivos de Semana Santa, que contará con 3 semanas), con 4 días de holgura en la última semana; finalmente, ya fuera de la empresa, se emplearán algunos de los días disponibles antes del depósito del trabajo para finalizar aspectos de seguimiento y conclusiones, así como para la revisión de la memoria.

Como se ha mencionado, un sprint tendrá una extensión de dos semanas. La duración aproximada será de 40 horas de trabajo, distribuidas en 4 horas al día, 5 días a la semana. Por último, el fin de cada sprint y comienzo del siguiente tendrá lugar en lunes.

1.6.1 Descomposición en tareas

En las siguientes tablas se muestra la descomposición en tareas, separadas según las componentes en que se divide el producto final.

Teniendo en cuenta que los objetivos principales del desarrollo están bastante bien concretados desde el principio, se definirán las tareas con un grado de precisión relativamente alto. Aun así, no se espera que estas tareas funcionen como unidades de trabajo finales, sino que en la etapa de desarrollo se incluirán dentro de historias que se adecuen mejor a la metodología.

Las Tablas 1, 2 y 3 listan aquellas tareas que son imprescindibles para el producto final del desarrollo, mientras que la Tabla 4 muestra las tareas adicionales que tendrán una prioridad más baja.

Tareas relativas al API Python		
Código	Título	Descripción
TA-01	Análisis del API	Analizar requisitos y casos de uso del API
TA-02	Diseño del API	Diseñar las clases, métodos y pruebas del API
TA-03	Creación de servidor GATT	Implementar la creación de un servidor GATT
TA-04	Conexión BLE	Implementar la conexión BLE entre cliente y servidor GATT
TA-05	Protocolo de comunicación	Implementar el protocolo a utilizar en la conexión BLE
TA-06	Capa de seguridad	Implementar la capa de seguridad sobre la interfaz BLE
TA-07	Pruebas del API	Realizar las pruebas unitarias y de integración diseñadas

Tabla 1. Lista de tareas relacionadas con el API Python del SDK

Tareas relativas a la aplicación Python		
Código	Título	Descripción
TP-01	Análisis de la aplicación	Analizar requisitos y casos de uso de la aplicación
TP-02	Diseño de la aplicación	Diseñar las clases, métodos y pruebas de la aplicación
TP-03	Interfaz BLE	Implementar los servicios y la comunicación BLE
TP-04	Fichero de propiedades	Implementar la comunicación con el fichero de propiedades
TA-05	Pruebas de la aplicación	Realizar las pruebas unitarias y de integración diseñadas

Tabla 2. Lista de tareas relacionadas con la aplicación Python de ejemplo

Tareas relativas a la aplicación móvil multiplataforma		
Código	Título	Descripción
TM-01	Análisis de la aplicación móvil	Analizar requisitos y casos de uso de la aplicación móvil
TM-02	Diseño de la aplicación móvil	Diseñar las clases, métodos, pruebas y prototipos de la aplicación móvil
TM-03	Interfaz BLE	Implementar el escaneo y comunicación con otros dispositivos ConnectCore por medio de BLE
TM-04	Interfaz gráfica	Implementar la interfaz gráfica de la aplicación móvil
TM-05	Pruebas de la aplicación móvil	Realizar las pruebas unitarias y de integración diseñadas

Tabla 3. Lista de tareas relacionadas con la aplicación móvil multiplataforma

Tareas extra		
Código	Título	Descripción
TE-01	Librería del proyecto	Extraer clases y métodos de la librería XBee que hayan sido utilizadas en el desarrollo a otra librería nueva.
TE-02	Mejora de la aplicación móvil	Mejorar la interfaz gráfica de la aplicación móvil y sus funcionalidades

Tabla 4. Lista de tareas adicionales de baja prioridad

1.6.2 Estimación de tiempos

En la Tabla 5 se recogen las estimaciones de tiempo de cada una de las tareas expuestas anteriormente, teniendo en cuenta no solo el tiempo empleado en el desarrollo, sino también en su documentación, y también se incluyen los tiempos dedicados tanto a la gestión del proyecto como a la elaboración del resto de apartados de la memoria. El principal criterio para realizar la estimación presentada es la complejidad que se espera que tengan las tareas una vez se trabaje con ellas.

Gestión		**25 h**
	Planificación	20 h
	Seguimiento	5 h
API Python		**120 h**
TA-01	Análisis del API	5 h
TA-02	Diseño del API	15 h
TA-03	Creación de servidor GATT	15 h
TA-04	Conexión BLE	30 h
TA-05	Protocolo de comunicación	15 h
TA-06	Capa de seguridad	25 h
TA-07	Pruebas del API	15 h
Aplicación Python		**60 h**
TP-01	Análisis de la aplicación	5 h
TP-02	Diseño de la aplicación	10 h
TP-03	Interfaz BLE	20 h
TP-04	Fichero de propiedades	15 h
TA-05	Pruebas de la aplicación	10 h
Aplicación móvil		**60 h**
TM-01	Análisis de la aplicación móvil	5 h
TM-02	Diseño de la aplicación móvil	10 h
TM-03	Interfaz BLE	10 h
TM-04	Interfaz gráfica	25 h
TM-05	Pruebas de la aplicación móvil	10 h
Memoria (Introducción y Conclusiones)		**35 h**
TOTAL		**300 h**

Tabla 5. Estimación de tiempos de las tareas

Como ya se ha mencionado, una vez en la etapa de desarrollo, se distribuirán las tareas en historias y se les asignará prioridades según su relevancia. Al asignarles puntos de historia, se tendrán en cuenta los tiempos especificados en esta sección.

Para cerrar este apartado, solo queda considerar que es posible que durante la elaboración del proyecto se den desviaciones respecto del plan establecido. Si esto sucediera, nos podríamos encontrar con dos casos diferentes: que el tiempo sea insuficiente para realizar todas las tareas, o que todas sean completadas con tiempo de sobra.

Para tratar el primer caso, se trabajará primero con las tareas de mayor prioridad, que aporten más valor al producto, de modo que las tareas que potencialmente queden sin acabar no sean tan relevantes para el resultado final.

Si se da el segundo caso, se añadirán las tareas propuestas en la Tabla 4 como tareas adicionales, ya que son las que más baja prioridad tienen y no se han considerado en la estimación general.

Capítulo 2. Desarrollo

El capítulo 2 estará centrado en todos los aspectos relacionados con el desarrollo del proyecto, incluyendo análisis, diseño, implementación y pruebas de los sistemas expuestos anteriormente.

Al tratarse de un desarrollo ágil basado en una metodología derivada de Scrum, todo el trabajo se repartirá a lo largo de una serie de sprints. Para el reparto se elaborarán historias de usuario partiendo de las tareas definidas en la planificación.

Dada la estructura del producto, se ha tomado la decisión de prescindir de una fase inicial de análisis y diseño global. En su lugar, se incorporan ambos como tareas en la pila de producto, de tal forma que al comenzar el desarrollo de cada componente del SDK tenga lugar el análisis y diseño únicamente de la parte correspondiente.

Justo antes de comenzar el primer sprint, se han concretado las historias y tareas que formarán parte de la pila de producto en el desarrollo de la primera componente, el API Python. En la Tabla 6 se muestran estas historias asociadas con las tareas de la planificación a las que engloban, junto con la estimación en puntos de historia. Un punto de historia equivaldrá a un día ideal de trabajo de 4 horas.

Código	Resumen	Tareas	Descripción	Puntos de historia
CCBLESDK-4	Análisis y diseño del API	TA-01, TA-02	Requisitos, casos de uso, diseño de clases y de casos de prueba.	5
CCBLESDK-5	GATT server	TA-03	Como desarrollador, quiero que el API me dé la opción de crear un servidor GATT, para poder comunicarme con otros dispositivos BLE.	3
CCBLESDK-6	BLE service	TA-04	Como desarrollador, quiero tener un servicio de ConnectCore BLE, para poder utilizarlo al crear cualquier aplicación para interactuar con dispositivos externos sobre BLE.	5
CCBLESDK-7	BLE connection	TA-04	Como desarrollador, quiero que el API maneje la conexión BLE, para ser notificado cuando se establece una nueva conexión.	3
CCBLESDK-8	Security layer	TA-06	Como desarrollador, quiero que el API tenga el mismo protocolo de seguridad al implantado en los XBees, para que mi conexión sea segura y los datos transmitidos estén encriptados.	8
CCBLESDK-9	Prueba de componentes del API	TA-07	Ejecutar los casos de prueba definidos en la fase de diseño.	5
CCBLESDK-10	BLE communication	TA-05	Como desarrollador, quiero que el servicio BLE/API tenga varios métodos para comunicarse, para poder transmitir y recibir datos de dispositivos externos.	3

Tabla 6. Tareas e historias correspondientes al API Python

Las tareas definidas siguen un orden secuencial, lo que significa que aquellas que están más abajo en la tabla dependen de que las que están por encima hayan sido completadas.

Como ya se concretó en la sección 1.6 Planificación inicial, los sprints tendrán una duración de 40 horas de trabajo, de modo que se elegirán tareas según correspondan con dicho número, tomando también en consideración el tiempo necesario para documentarlas.

Por todo ello, en la planificación de cada sprint se repartirán 32 horas (8 puntos de historia) entre las tareas, y se utilizarán las 8 horas restantes para trabajar en la documentación de la memoria.

2.1 Sprint 1 (24 feb. – 9 mar.)

2.1.1 Planificación del sprint

Las tareas e historias que compondrán el primer sprint quedan recogidas en la Tabla 7 .

Historia	Resumen	Puntos de historia	Horas
CCBLESDK-4	Análisis y diseño del API	5	20
CCBLESDK-5	GATT server	3	12

Tabla 7. Tareas asignadas al sprint 1

Si se completaran ambas antes de dar por concluido el sprint, la siguiente tarea elegida sería CCBLESDK-6.

La primera tarea por realizar es **CCBLESDK-4**, la cual comprende tanto el análisis como el diseño de toda el API Python. Todo esto queda documentado a continuación en las secciones 2.1.2 Análisis y 2.1.3 Diseño.

2.1.2 Análisis del API

A partir de la información reunida durante la planificación del proyecto, y especialmente durante reuniones con los tutores acerca de las especificaciones del sistema, se elabora el análisis del API.

Tras esto quedarán perfectamente definidos los requisitos que ha de cumplir para que el producto sea aceptado, y los posibles casos de uso y roles de usuario que debemos considerar.

2.1.2.1 Identificación de los usuarios del API

En el uso del API se ha detectado como posible rol que puede asumir el usuario el de “desarrollador”, quien utilizará el API para desarrollar aplicaciones en Python para ConnectCore.

En un contexto real, una vez se ponga en explotación, este rol será tomado tanto por desarrolladores internos de Digi como por sus clientes que deseen definir su propio protocolo en la comunicación con un ConnectCore a través de la interfaz BLE.

2.1.2.2 Catálogo de requisitos del API

Los requisitos funcionales que debe satisfacer el API se muestran en la Tabla 8.

Código	Requisito
R-01	El API distinguirá si el ConnectCore en el que se encuentra tiene soporte BLE nativo.
R-02	El API distinguirá si el ConnectCore en el que se encuentra tiene conectado un XBee.
R-03	El API podrá comprobar si el XBee conectado tiene Bluetooth activado.
R-04	Si el ConnectCore tiene soporte BLE nativo y tiene un XBee conectado, se utilizará la interfaz BLE nativa por defecto.
R-05	El API podrá crear un servidor GATT, siempre con los mismos servicios y características, iguales a los del servidor GATT de un XBee.
R-06	El API permitirá al ConnectCore anunciar el servicio y detenerlo.
R-07	El API ofrecerá al usuario la posibilidad de customizar el nombre anunciado.
R-08	El API permitirá al ConnectCore conectarse con un cliente BLE.
R-09	El API permitirá mantener comunicación entre el ConnectCore y el cliente por medio del protocolo definido en el API de XBee.
R-10	Al iniciar la comunicación, el API exigirá al usuario que se autentique para poder proseguirla.
R-11	El API permitirá enviar datos a través de la característica de lectura durante la comunicación.
R-12	El API permitirá al cliente suscribirse a la característica por la que se envíen los datos.
R-13	El API podrá recibir datos entrantes a través de la característica de escritura y reaccionar a ellos.
R-14	El API podrá distinguir si un cliente está conectado o no.

Tabla 8. Requisitos funcionales del API

El cumplimiento de todos estos requisitos tiene prioridad máxima, pues se refieren a la funcionalidad básica que debe tener el API.

Por otra parte, el API también tiene una serie de requisitos no funcionales que debe satisfacer igualmente.

- **Requisitos tecnológicos**: es importante recalcar que debe funcionar correctamente una vez utilizado en un ConnectCore 6UL, ya sea con BLE nativo o mediante la interfaz de un XBee 3. Además, para ConnectCore y para todas las variantes de XBee que no tienen antena integrada, no será posible la conexión BLE sin acoplar antenas en los conectores U.FL o RP-SMA[4] disponibles.
- **Requisitos de seguridad**: el API utilizará los protocolos SRP para la autenticación y AES-256 para encriptar los datos, con el fin de asegurar la seguridad en la comunicación. El principal motivo en la elección de estos protocolos es el hecho de que son los utilizados en el ecosistema XBee.
- **Requisitos de usabilidad**: cuando el usuario utilice el API para sus propios desarrollos, la interfaz Bluetooth que utilice el ConnectCore (nativa o XBee) deberá ser transparente;

[4] Ambos se tratan de tipos de conectores presentes en los ConnectCore o XBee.
Véanse: *"U.FL and W.FL Series Connectors - Hirose | DigiKey"* y RP-SMA Coax Connectors

deberá aportar la misma funcionalidad independientemente de cuál se use. Del mismo modo, la capa de seguridad será totalmente transparente al usuario.

- **Requisitos de fiabilidad**: si el API no fuera capaz de detectar ninguna interfaz BLE disponible, o si sucede algún problema en la conexión, deberá propagar la excepción correspondiente al usuario.

2.1.2.3 Casos de uso del API

En la Figura 3 se expone el diagrama de casos de uso del API. Se considera el rol de desarrollador mencionado en la sección 2.1.2.1 como el único actor.

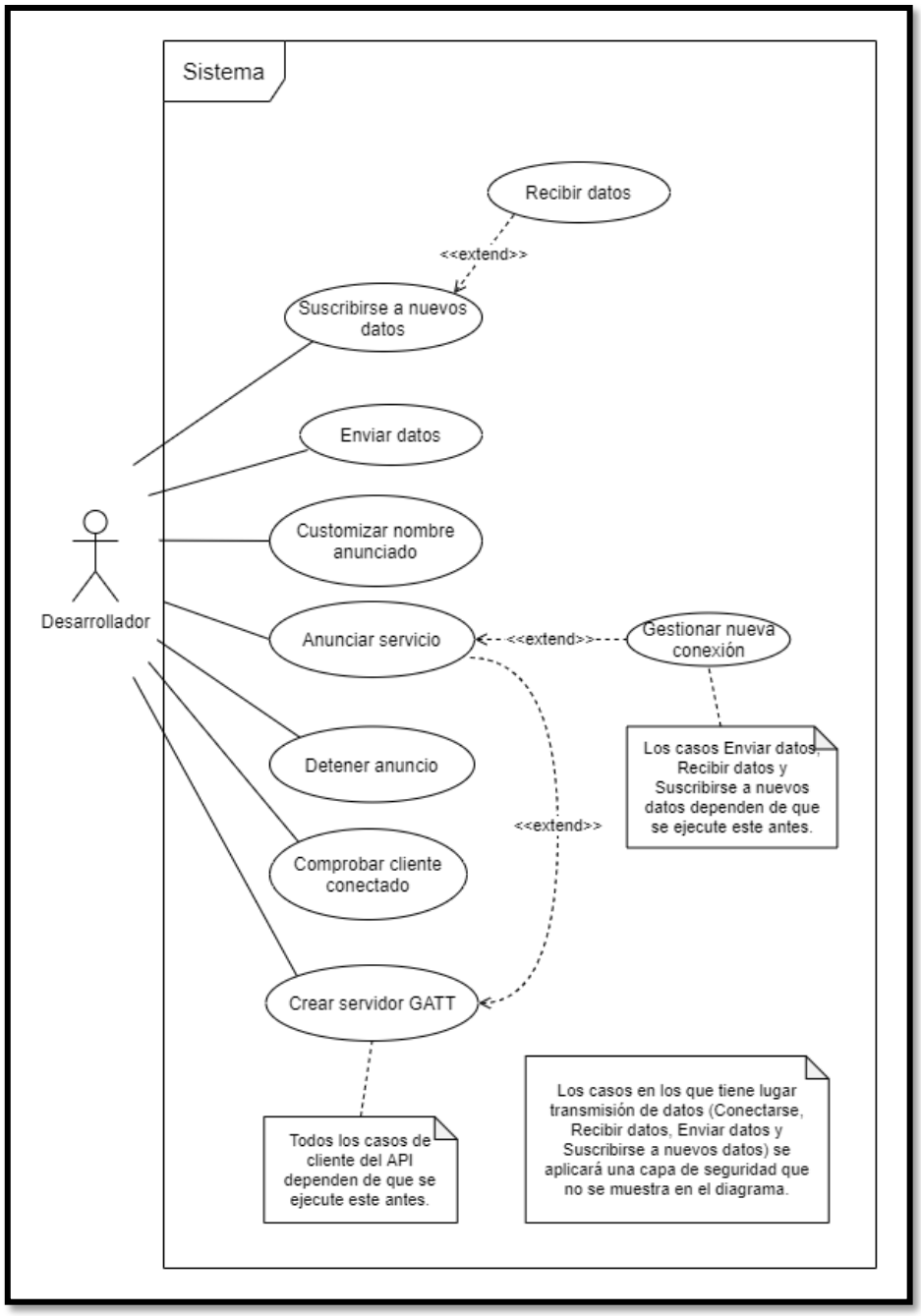

Figura 3. Diagrama de casos de uso del API

Se barajó la opción de incluir dos actores, uno correspondiente a una aplicación que utilizara el API y un cliente de dicha aplicación, para ofrecer un punto de vista más global del funcionamiento del API, pero se optó por la opción elegida por ser más específico para el uso real del API.

2.1.3 Diseño del API

Los diagramas de las Figuras 4, 5 y 6 muestran el diseño de las clases que seguirá la implementación final del API. Los tres juntos forman un único diagrama, pero ya que resultaba fácil establecer una división lógica del API en tres partes, se ha tomado la decisión de separarlo por ser demasiado extenso y algo confuso.

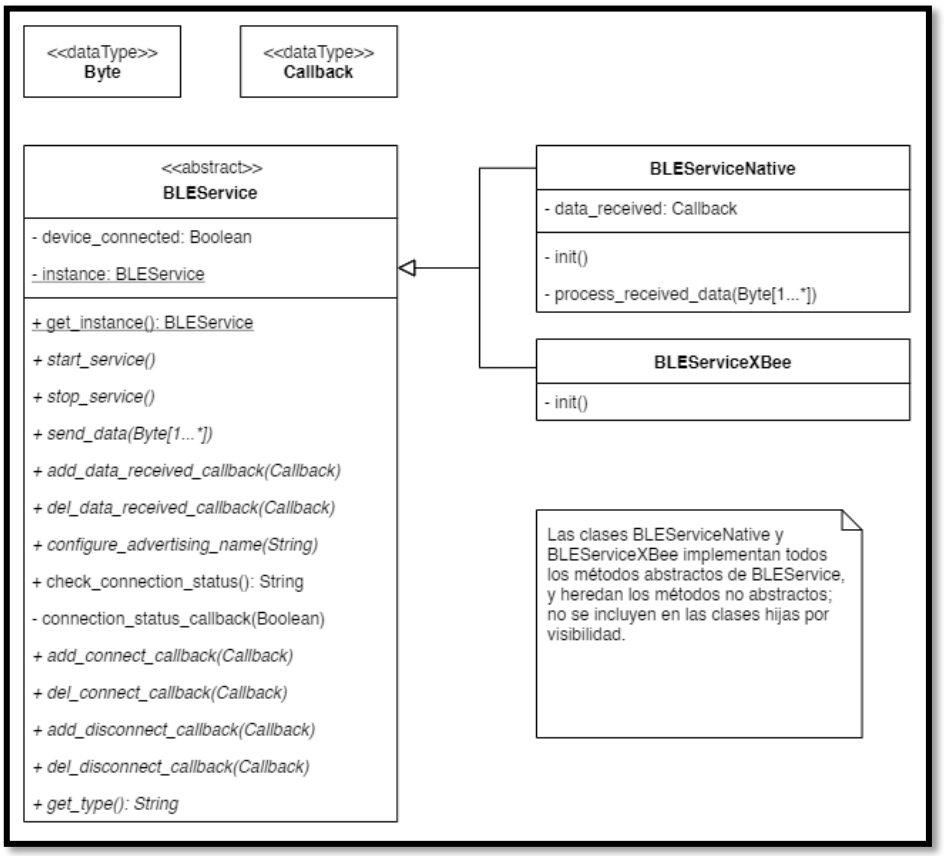

Figura 4. Diagrama de clases del API. Herencia entre BLEService y sus hijas

Así, la Figura 4 muestra el servicio principal BLEService, que es la clase principal y la que estará disponible para el usuario desarrollador final. Esta clase abstracta proporciona a través de sus métodos el acceso a las principales funcionalidades del sistema, y abstrae la forma en que está implementada la interfaz Bluetooth y la seguridad por medio de sus dos clases hijas, BLEServiceNative y BLEServiceXBee. La clase BLEService solo podrá tener una instancia al mismo tiempo, ya sea suya o de cualquiera de sus hijas, y para ello se aplicará el patrón ***singleton***.

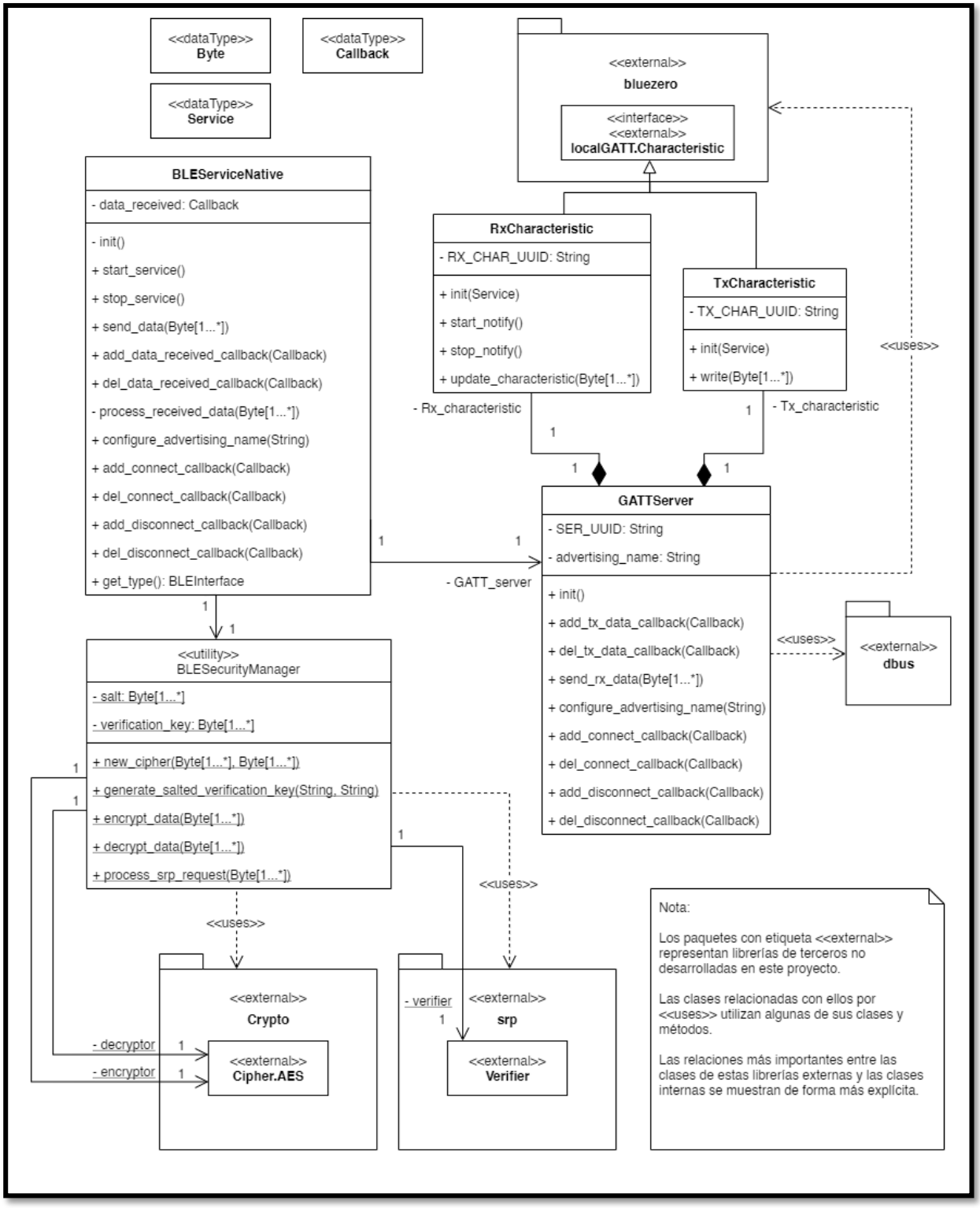

Figura 5. Diagrama de clases del API. Clases para la implementación de BLEService con la interfaz nativa

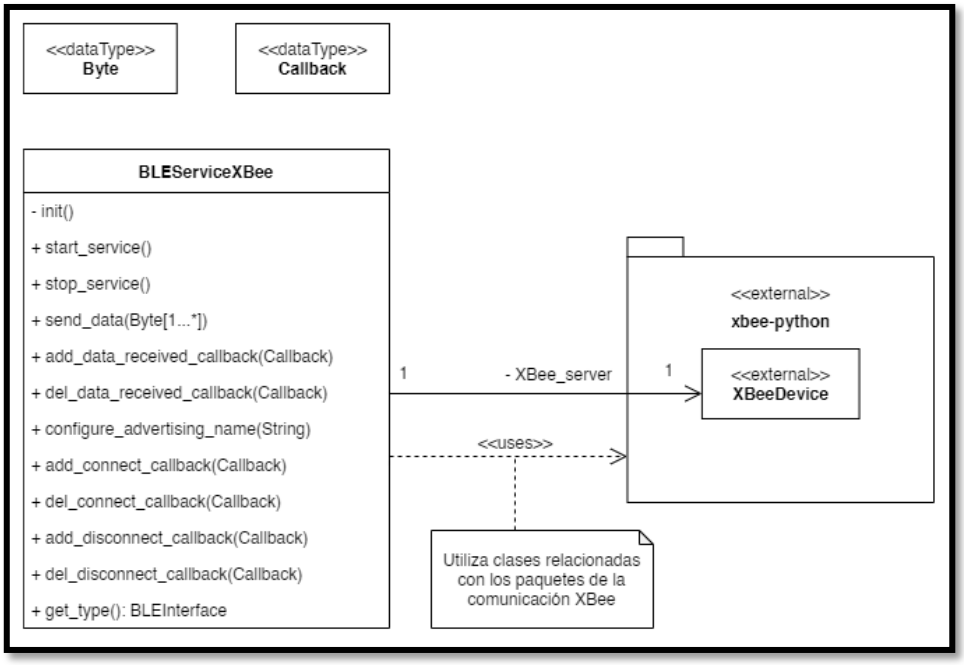

Figura 6. Diagrama de clases del API. Clases para la implementación de BLEService con la interfaz XBee

La principal diferencia en la implementación del servicio para cada interfaz proviene del hecho de que muchas de las funciones necesarias para el API ya están implementadas para el XBee, mientras que no es así en el caso del ConnectCore.

Por ello, al utilizar la interfaz nativa será necesario crear un servidor GATT, representado por la clase GATTServer, que se ocupe de gestionar la comunicación con el dispositivo al que esté conectado por medio de sus características de escritura y lectura.

También será necesario gestionar la seguridad (el cifrado de datos para una conexión segura), por medio de la clase BLESecurityManager, teniendo en cuenta además que como esta capa de seguridad ya existe en xbee-python, la nueva implementación deberá funcionar de forma muy similar.

Estas dos clases mencionadas se explicarán en mayor detalle en el sprint en que se vayan a desarrollar.

A continuación, se explican brevemente los métodos de BLEService y cómo estarán implementados según la interfaz empleada.

get_instance: obtiene la única instancia de clase según define el patón *singleton.* La clase instanciada no será BLEService, sino que el método determinará la interfaz BLE que esté disponible y creará una instancia de la clase correspondiente.

start_service: comienza el servicio Bluetooth.

- BLE nativo: crea un GATTServer si no existe ya, e inicia el anuncio de sus servicios.
- BLE XBee: inicia la comunicación con el XBeeDevice que se encuentra en el puerto serie del ConnectCore para intercambiar datos con su servidor GATT, activa la interfaz bluetooth y automáticamente inicia el anuncio.

stop_service: detiene el servicio Bluetooth.

- BLE nativo: detendrá el anuncio de un GATTServer en funcionamiento.
- BLE XBee: inhabilitará la interfaz bluetooth, según especifique el método disable_bluetooth de la librería xbee-python.

send_data: envía datos a través de la interfaz bluetooth activa.

- BLE nativo: envía los datos al GATTServer, que a su vez utiliza send_Rx_data para pasarlos a su RxCharacteristic. Esta se encargará de actualizar su valor y de notificar el cambio. Los datos serán empaquetados a través de la librería xbee-python y serán cifrados por BLESecurityManager.
- BLE XBee: envía los datos a través del XBeeDevice, que se encarga de empaquetarlos y de gestionar la seguridad.

add_data_received_callback: recibe una función callback a la que se llamará cada vez que se reciban nuevos datos desde la interfaz bluetooth. Para eliminar esa llamada se utiliza **del_data_received_callback**.

- BLE nativo: guarda en la variable **data_recieved** la función de callback a la que se llamará cada vez que se reciban nuevos datos, y pasa el método process_recieved_data, que gestiona aspectos de seguridad y comunicación, como callback al servidor GATT.
- BLE XBee: suscribe al método add_bluetooth_data_received_callback, de forma que se ejecute la función cada vez que reciba datos. La seguridad y el formateo de datos se gestionan automáticamente.

process_received_data: solo en la interfaz nativa, desencripta y desempaqueta los datos que recibe, y ejecuta la función callback en **data_recieved.**

configure_advertising_name: cambia el nombre que mostrará el servicio al anunciarse.

- BLE nativo: el GATTServer cambia el nombre con configure_advertising_name.
- BLE XBee: no existen métodos que abstraigan esta funcionalidad aún, así que hace una petición al XBeeDevice pidiendo el cambio del nombre anunciado.

check_connection_status: para ambas interfaces devuelve la información acerca de la comunicación contenida en la variable **device_connected**.

connection_status_callback: en las dos interfaces, es el callback por defecto al que se llama cuando tiene lugar una conexión o desconexión de un cliente BLE, y almacena esos datos en device_connected.

add_connect_callback y add_disconnect_callback: reciben una función callback a la que se llamará cada vez que se establezca y finalice respectivamente una conexión BLE con el cliente. Para eliminar estas llamadas se utilizan **del_connect_callback y del_disconnect_callback**.

- BLE nativo: suscribe a funciones del D-Bus[5] en el GATTServer para recibir información de los cambios que se puedan producir en la conexión con el dispositivo cliente.

[5] El D-Bus es un sistema para la comunicación y notificación entre distintos procesos, al cual se pueden conectar aplicaciones que deseen enviarse mensajes entre ellas o al sistema operativo. Por ejemplo, en este proyecto se utiliza para comunicar el servidor GATT con el adaptador bluetooth y con los dispositivos conectados por medio de BLE.

- BLE XBee: suscribe al método add_modem_status_received_callback del XBeeDevice para obtener de éste la información del estado de la conexión.

get_type: devuelve información sobre el tipo de interfaz utilizada por la instancia en un formato legible para humanos. Aunque sería posible obtener esta información mediante funciones que detecten el tipo de una instancia en tiempo real, esto exigiría hacer referencia a las clases BLEServiceNative y BLEServiceXBee, y se supone que dichas clases no deben aparecer ante el usuario final, puesto que se rompería la capa de abstracción que provee BLEService.

2.1.4 Diseño de las pruebas del API

En esta sección se definen las pruebas a las que se someterá la componente API del proyecto una vez terminada.

Como parte de la última tarea de la componente, se incluirá una sección en la que se indicará una por una si ha sido posible superar la prueba o no.

Código	Prueba
PA-01	Comprobar con una aplicación estándar de escaneo de dispositivos BLE que el servidor anunciado es visible desde interfaz nativa o XBee, y que el cliente las ve de igual forma.
PA-02	Comprobar con una aplicación estándar de escaneo de dispositivos BLE que no sea posible establecer la comunicación sin haber completado el proceso de autenticación.
PA-03	Comprobar con una aplicación estándar de escaneo de dispositivos BLE que tras establecer conexión el cliente puede ver las características definidas independientemente de la interfaz BLE usada.
PA-04	Comprobar que al detener el servicio bluetooth el servidor ya no sea visible.
PA-05	Comprobar con una aplicación de escaneo de dispositivos BLE que implemente los protocolos de seguridad que los datos se envían correctamente.
PA-06	Comprobar con una aplicación de escaneo de dispositivos BLE que implemente los protocolos de seguridad que los datos se reciben correctamente.
PA-07	Comprobar que el API detecta cuándo hay un dispositivo conectado y cuándo no.
PA-08	Comprobar que tras cambiar el nombre del servicio anunciado una aplicación de escaneo de dispositivos BLE detecte el servidor con el nuevo nombre.

Tabla 9. Diseño de pruebas del API

A partir de las siguientes secciones se tratarán todos los aspectos importantes de la implementación del API, incluyendo tanto el código generado como los problemas que surjan en el proceso. Para organizarlo, se divide en distintos subapartados correspondientes a cada una de las historias de implementación que hayan sido completadas en el sprint.

2.1.5 CCBLESDK-5 – GATT Server

Al tratarse de la primera tarea de implementación, se ha incluido como subtarea preparar el entorno de desarrollo en Pycharm.

En primer lugar, se genera un fichero requirements.txt en el que se incluyen todos los paquetes necesarios para el proyecto. Pycharm permite instalar automáticamente todo lo contenido en este fichero.

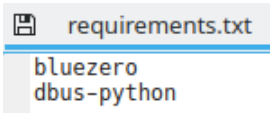

Figura 7. requirements.txt al inicio del proyecto

De momento, los paquetes necesarios son *bluezero*, que gestiona todo lo relacionado con Bluetooth Low Energy en la aplicación, y *dbus-python*, que gestiona el sistema D-Bus, que posee un demonio que funciona como bus para transmitir mensajes enviados a las características del servidor GATT.

En un entorno de desarrollo Python es recomendable trabajar con un intérprete Python en un entorno virtual, dado que de esta forma todos los paquetes con los que se trabaja en el proyecto son almacenados en una carpeta site-packages dentro del intérprete Python de dicho entorno. El propio Pycharm permite configurarlo de forma sencilla en Settings, e instalar en él fácilmente los paquetes necesarios.

El primer problema del desarrollo apareció al intentar preparar el entorno virtual, dado que una vez creado, no permitía instalar correctamente los paquetes. Se intentó eliminarlo y volver a crearlo, pero en el nuevo, aunque permitía la instalación, guardaba los paquetes en el intérprete general de Python. La solución final fue reiniciar el equipo, dado que la documentación de Pycharm mencionaba que ese paso era necesario para activar el entorno virtual creado.

Una vez listo el proyecto, se creó el fichero **GATTServer.py**, que contiene las clases GATTServer, RxCharacteristic y TxCharacteristic.

RxCharacteristic y **TxCharacteristic** sirven como interfaz de comunicación con un cliente externo. El primero permite al cliente suscribirse a notificaciones de nuevos datos que ofrezca el servidor GATT, y el segundo recibe los datos enviados por el cliente y los transmite hacia el servidor.

- **GATTServer** contiene como atributos objetos de las dos clases anteriores, y se ocupa de gestionarlas, de establecer la comunicación con el D-Bus, de iniciar el anuncio del servicio y de ofrecer todas las funcionalidades necesarias para que un tercero que utilice la clase pueda enviar y recibir datos a través suyo.

A la hora de documentar los métodos en el código, se ha optado por seguir el formato de Docstring de Google, con el fin de poder generar documentación fácilmente. Se ha elegido este formato por ser el utilizado en otros proyectos de Digi, y porque Pycharm ofrece la opción de escribir con él de forma sencilla.

Por último, en esta tarea también se creó el repositorio en Git en el que se almacenará el proyecto, y se realizó un commit tras terminar de implementar el GATTServer.

2.1.6 Revisión del sprint

La información sobre las horas dedicadas a cada tarea se recoge en la Tabla 10.

Tarea	Horas estimadas	Horas reales	Desviación
Análisis y diseño del API	20	30	+50 %
GATT server	12	8	-33,3 %
Documentación	8	2	-50 %

Tabla 10. Comparativa de horas dedicadas en el sprint 1

La Figura 8 muestra la evolución del sprint, y en ella se puede ver un resumen visual del tiempo que se ha tardado en completar las historias.

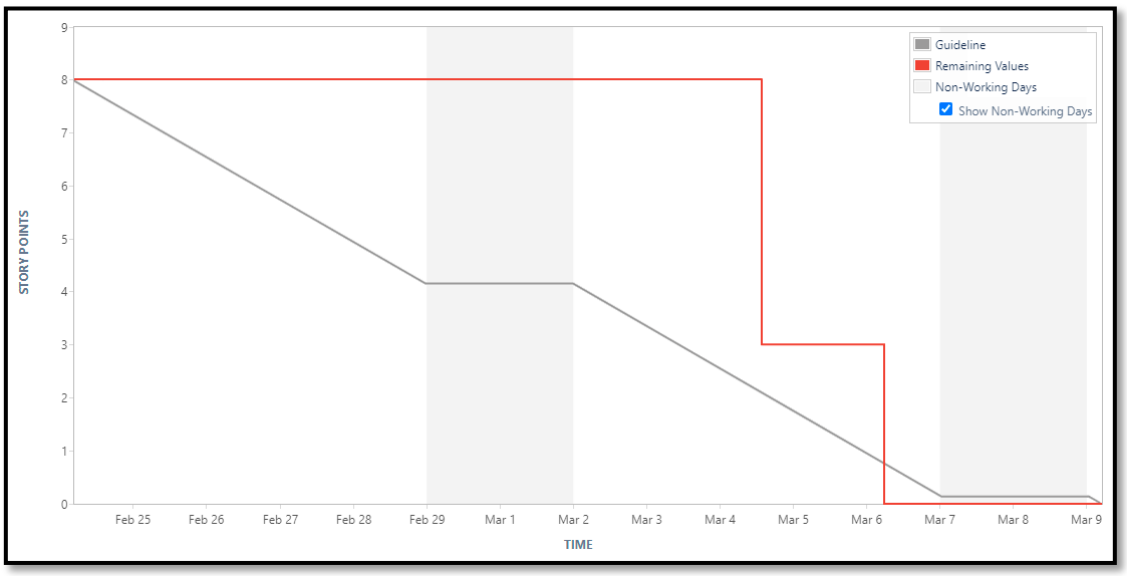

Figura 8. Burndown chart del sprint 1

Las dos historias fueron completadas correctamente a tiempo, y no hubo especiales complicaciones. Aun así, se puede apreciar cómo la primera tarea, análisis y diseño, ha llevado mucho más tiempo que la segunda y también más de lo que se pretendía en primer lugar.

La razón de esto es una mala estimación del tiempo que podía tomar una tarea que comprendía todo el análisis y diseño iniciales junto con todo el trabajo extra que suponía registrarlo en la memoria. Por ello, ya que habrá otras tareas similares al avanzar a nuevas fases del proyecto, se tendrá en cuenta lo aprendido en esta revisión.

2.2 Sprint 2 (9 mar. – 23 mar.)

De nuevo, el primer día del periodo se han seleccionado las tareas que van a formar parte del sprint.

Historia	Resumen	Puntos de historia	Horas
CCBLESDK-6	BLE service	5	20
CCBLESDK-7	BLE connection	3	12

Tabla 11. Tareas asignadas al sprint 2

Al igual que en el anterior, se ha elegido las historias en base al orden en que deben realizarse por depender unas de otras. Si hubiera tiempo extra, la siguiente tarea será CCBLESDK-8, Security Layer.

2.2.1 CCBLESDK-6 – BLE service

En esta tarea se ha creado el fichero BLEService.py, en el que se incluyen las clases BLEService, BLEServiceNative y BLEServiceXBee.

BLEService es la clase que proporciona los métodos que estarán disponibles para ser utilizados por el usuario final del API. Es una clase abstracta, y para ello hace uso del módulo abc de Python, que permite utilizar la etiqueta @abstractmethod para calificar a los métodos de abstractos. Sus métodos y propiedades más relevantes implementados hasta ahora son los siguientes:

- **get_instance**: comprueba las interfaces bluetooth disponibles en el dispositivo y genera una única instancia de la clase, según dice el patrón Singleton. Esa instancia puede ser de tipo BLEServiceNative o BLEServiceXBee según la interfaz detectada. Lanza una excepción BluetoothNotSupportedException si no encuentra ninguna.
- A la hora de comprobar la interfaz bluetooth de un XBee conectado al dispositivo, el método trata de crear una conexión de prueba en el puerto "/dev/ttyXBEE", que es donde se conectan estos módulos, y con diferentes tasas de baudios[6] estándar.
- **_BLE_interface**: guarda el tipo de interfaz bluetooth utilizada. Para ello hace uso de la enumeración **BLEInterface** creada en esta historia en BLEInterface.py, que puede tomar como valores NATIVE_INTERFACE y XBEE_INTERFACE.

Además, en BLEService se han declarado todos los métodos considerados durante la planificación a modo de esqueleto.

BLEServiceNative implementa los métodos abstractos de BLEService comunicándose con el GATTServer que crea. Sus métodos y propiedades más relevantes implementados hasta ahora son los siguientes:

[6] La tasa de baudios (en inglés baud rate) es el número de unidades de señal por segundo que se envían en una comunicación. Para que la conexión se dé lugar y dos dispositivos se entiendan entre sí, deben utilizar la misma tasa de baudios.

- **start_service**: llama al método start del GATTServer para iniciar su servicio. Para conseguir que el método funcione correctamente sin bloquear la ejecución del programa, se han hecho ligeras modificaciones para ejecutarlo en un hilo diferente por medio del módulo threading de Python.
- **stop_service**: llama al método stop del GATTServer, que detiene la ejecución de la aplicación del servidor y al hilo.
- **configure_advertising_name**: llama al método del GATTServer que cambia el alias del adaptador.

BLEServiceXBEE implementa los métodos abstractos de BLEService a partir del servidor GATT ya presente en un XBee. Sus métodos y propiedades más relevantes implementados hasta ahora son los siguientes:

- **start_service** y **stop_service** habilitan y deshabilitan la interfaz bluetooth del XBee.
- **configure_advertising_name**: la librería xbee-python no ofrece métodos para hacerlo directamente, así que se modifica mediante el método genérico de cambio de parámetros el parámetro BI presente en el XBee, que corresponde al nombre del adaptador.

También se ha creado el fichero **exception.py**, que contiene las excepciones personalizadas que puedan ocurrir. Las excepciones definidas hasta ahora son ConnectCoreBLEException, la excepción genérica, y BluetoothNotSupportedException, que hereda de ConnectCoreBLEException.

Otro fichero creado ha sido **utils.py**, en el que se esperan guardar las funciones básicas que se pueden utilizar en todo el API. La única función definida hasta ahora es doc_enum, traída desde la librería xbee-python, que genera documentación a partir de las descripciones de los valores de una enumeración.

2.2.2 CCBLESDK-7 – BLE connection

En esta historia se han implementado algunos de los métodos de BLEService y sus clases hijas, y se han hecho modificaciones a GATTServer para permitir reconocer cuándo se producen conexiones y desconexiones.

En BLEService se han añadido dos propiedades privadas **_on_connect** y **_on_disconnect**, que se tratan de listas que contienen funciones a ser llamadas cuando se reconoce el evento de conexión o desconexión respectivo a cada interfaz.

Tanto al inicializar BLEServiceNative como BLEServiceXBee, se añade un método manejador del evento de conexión específico de cada interfaz. El método manejador en ambos casos es el método privado _execute_user_connection_callbacks, que llama a todos los métodos de las listas mencionadas en el párrafo anterior, dependiendo de si recibe información de que se ha producido una conexión o una desconexión.

- En el caso de **BLEServiceXBee**, para implementar el proceso se ha utilizado la infraestructura de paquetes y mensajes utilizados en el XBee, ya que al producirse una conexión o desconexión BLE el módulo envía un mensaje de tipo ModemStatus con información que lo detalla.
- En el caso de **BLEServiceNative**, fue más problemático encontrar una solución. La librería utilizada para gestionar los procesos de bluetooth, bluezero, no parecía ofrecer

ninguna forma de suscribirse a los eventos de conexión. Por ello, fue necesario acceder directamente al D-Bus desde el GATTServer.

Finalmente, la estrategia seguida consiste en obtener todos los objetos gestionados por el D-Bus (incluyendo escuchar la posible llegada de nuevos objetos), y suscribir cada uno de ellos a la señal PropertiesChanged que se envía cuando cambia una propiedad. Cada vez que se recibe esa señal, se ejecuta un método manejador que detecta si el cambio se ha producido en la propiedad "Connected" , y si es así, ejecuta los métodos de callback que se reciben desde BLEServiceNative.

Por último, también se ha implementado un método para obtener el estado de la conexión sin tener que recurrir a funciones de callback. En BLEServiceXBee simplemente se guarda la información recibida de los callbacks en una variable que luego se devuelve.

En BLEServiceNative, las pruebas hicieron ver que la señal escuchada no siempre se emite si el dispositivo conectado no está enlazado correctamente, así que para implementar el método se accede de nuevo al D-Bus para comprobar la información de conexión directamente de los dispositivos gestionados.

2.2.3 Revisión del sprint

La información sobre las horas dedicadas a cada tarea se recoge en la Tabla 12.

Tarea	Horas estimadas	Horas reales	Desviación
BLE service	20	11	-45 %
BLE connection	12	16	+33,3 %
Documentación	8	5	-37,5 %

Tabla 12. Comparativa de horas dedicadas en el sprint 2

En la segunda semana de este sprint da comienzo la situación excepcional de cuarentena por el COVID-19. A partir de este momento y mientras dure, va a cambiar el flujo de trabajo mantenido hasta ahora.

Las reuniones más importantes con los tutores se mantienen y se llevan a cabo por videoconferencia, pero se vuelve más difícil aclarar cualquier duda encontrada durante el desarrollo.

Para poder continuar con el trabajo desde casa, he traído desde la oficina todos los recursos necesarios incluyendo un módulo ConnectCore, el cableado necesario y demás.

La Figura 9 muestra la evolución del sprint, y en ella se puede ver un resumen visual del tiempo que se ha tardado en completar las historias.

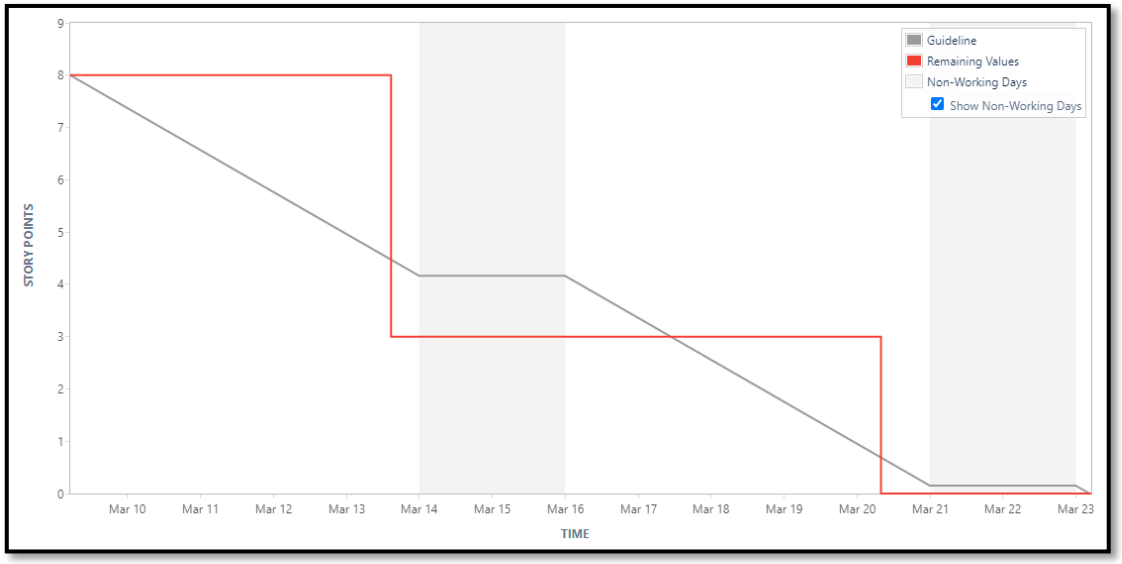

Figura 9. Burndown chart del sprint 2

Esta gráfica no es tan precisa como la anterior por distintos motivos. Primero, algunos días son perdidos en preparar el nuevo entorno de trabajo a distancia, incluyendo la disposición física de los recursos, la conexión a la VPN de la empresa, etc. Segundo, dado que dispongo de todo lo necesario para trabajar en cualquier momento, ya no existen restricciones respecto a horas de trabajo diarias. Por ello, se ha intentado recuperar el tiempo perdido en asuntos externos al proyecto invirtiendo algunas horas extra.

Finalmente, aún con todos los problemas encontrados, se ha logrado finalizar el sprint satisfactoriamente, y los siguientes hasta el final de la cuarentena deberían transcurrir con relativa normalidad al tener el entorno listo.

2.3 Sprint 3 (23 mar. – 6 abr.)

Para este sprint se han escogido las tareas contenidas en la Tabla 13.

Historia	Resumen	Puntos de historia	Horas
CCBLESDK-10	BLE communication	3	12
CCBLESDK-8 (Parte I)	Security Layer	5	20

Tabla 13. Tareas asignadas al sprint 3

Para comenzar CCBLESDK-8, es necesario haber completado antes CCBLESDK-10. La historia CCBLESDK-8 es algo más larga de lo habitual, pero debe completarse ya que es la última tarea de implementación restante en esta sección del proyecto. Si el tiempo no fuera suficiente, quedará como trabajo extra para el siguiente sprint.

2.3.1 CCBLESDK-10 – BLE communication

Esta historia se ha dividido en dos subtareas: una relativa al envío de datos a un cliente, y otra a la recepción de los datos. En ambos casos se han implementado métodos de la clase BLEService y sus clases hijas.

La única complejidad de esta tarea proviene de que, para comunicarse a través de Bluetooth, los XBee utilizan un formato de paquete específico definido en la librería XBee llamado User Data Relay (Figura 10).

Como el uso de ambas interfaces debe ser homogéneo para el usuario final, si la interfaz BLE XBee utiliza estos paquetes, también la interfaz BLE nativa debe hacerlo.

Para envío de datos, el único método que se utiliza es **send_data**, y su implementación varía según la interfaz utilizada:

- En BLEServiceNative, send_data recibe los datos a enviar, y se utiliza la clase UserDataRelayPacket de la librería XBee para empaquetarlos. Por último, se envían como vector de bytes a través del servidor GATT con send_rx_data. El servidor actualiza la característica de lectura y notifica el cambio.
- En BLEServiceXBee, basta con utilizar send_bluetooth_data(data) de la clase XBeeDevice para que se encargue de empaquetar los datos y enviarlos.

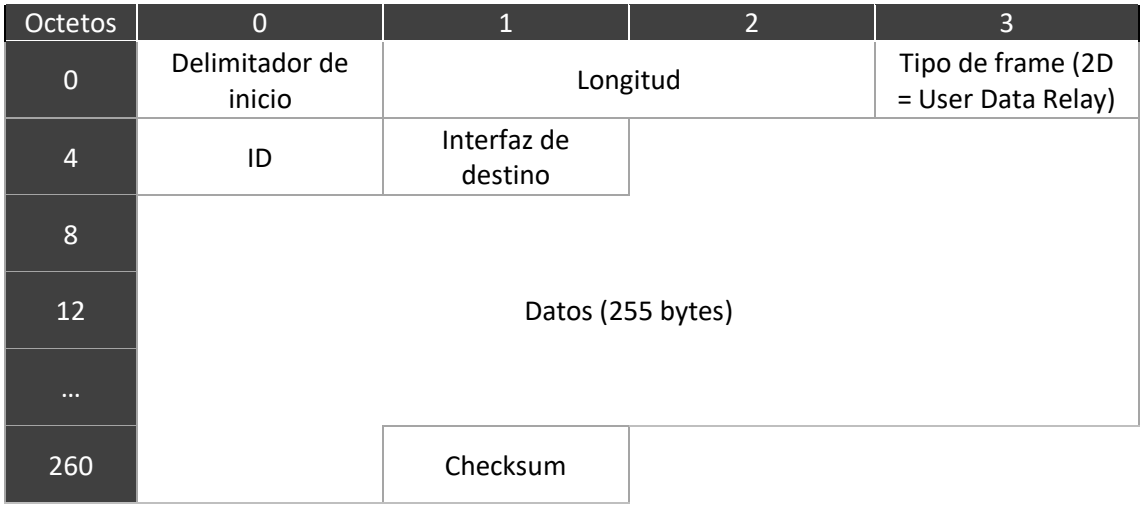

Figura 10. Formato del paquete User Data Relay

Para recibir datos, se utiliza la propiedad privada **_on_data_received**, que almacena todas las funciones a ser llamadas cuando se detectan nuevos datos. El método encargado de esas llamadas es **_execute_user_data_received_callbacks**, que recibe un paquete User Data Relay.

- En BLEServiceNative, _execute_user_data_received_callbacks extrae los datos útiles del paquete y llama a cada función de _on_data_received con dichos datos. El método se pasa al servidor GATT como la función a ser llamada si se reciben datos por la característica de escritura.

- En BLEServiceXBee, _execute_user_data_received_callbacks llama a cada función de _on_data_received con el paquete completo. El método se pasa a add_modem_status_received_callback del XBeeDevice, y siempre que reciba un paquete User Data Relay lo desempaquetará automáticamente.

2.3.2 CCBLESDK-8 – Security Layer (Parte I)

Se ha implementado una nueva clase estática, **BLESecurityManager**, que se encargará de gestionar todos los aspectos de autenticación y seguridad. Esta clase solo es necesaria al utilizar la interfaz BLE nativa del ConnectCore; los métodos de la librería XBee permiten gestionarla automáticamente.

Los métodos y propiedades más relevantes implementados son los siguientes:

- **new_cypher** crea un cifrador AES con modo contador, a partir de una clave de sesión y un número arbitrario calculado anteriormente (nonce).
- **encrypt_data** y **decrypt_data** encriptan y desencriptan datos respectivamente, cifradores obtenidos de new_cypher.
- **generate_salted_verification_key** genera la sal y la clave de verificación que serán utilizadas en la autenticación a través de una contraseña que se introduce como parámetro.
    - Con el fin de gestionar el cambio de contraseñas para la autenticación, se ha añadido a BLEService un nuevo método, **set_password**. En BLEServiceNative llama a generate_salted_verification_key con la nueva contraseña para generar la sal y la clave sin almacenarla. Si se llama en BLEServiceXBee, actualiza la contraseña mediante XBeeDevice.update_bluetooth_password.
- **process_srp_request** recibe paquetes de datos de tipo BLE Unlock API. Este tipo de paquete se utiliza para autenticar la conexión en la interfaz Bluetooth de un XBee. Contiene información de la fase de autenticación a la que corresponde, y de varios datos utilizados en el protocolo SRP y en el cifrador AES. El método se encarga de gestionar el proceso de autenticación SRP a partir de los datos del paquete, según la fase en la que se encuentre.

En los métodos de envío y recepción de datos de BLEServiceNative se han hecho modificaciones para incluir encriptación (al enviar los datos) y desencriptación (al recibir datos encriptados), y también se han añadido los casos en los que el paquete recibido es del tipo BLE Unlock API, para pasar la petición directamente a BLESecurityManager.

Al acabar la implementación se procede a realizar algunas pruebas básicas para determinar si funciona correctamente la autenticación y la encriptación de datos. Para estas pruebas, la empresa dispone de una aplicación, Digi Mobile, que permite la conexión con un dispositivo XBee (o un servidor GATT equivalente como puede ser el creado con BLEServiceNative).

El resultado es que no ha sido posible autenticarse correctamente, y como el tiempo no es suficiente, será necesario encontrar la causa de este problema en el siguiente sprint.

2.3.3 Revisión del sprint

La información sobre las horas dedicadas a cada tarea se recoge en la Tabla 14.

Tarea	Horas estimadas	Horas reales	Desviación
BLE communication	12	12	0 %
Security Layer	20	24	+20 %
Documentación	8	4	-50 %

Tabla 14. Comparativa de horas dedicadas en el sprint 3

La Figura 11 muestra la evolución del sprint, y en ella se puede ver un resumen visual del tiempo que se ha tardado en completar las historias.

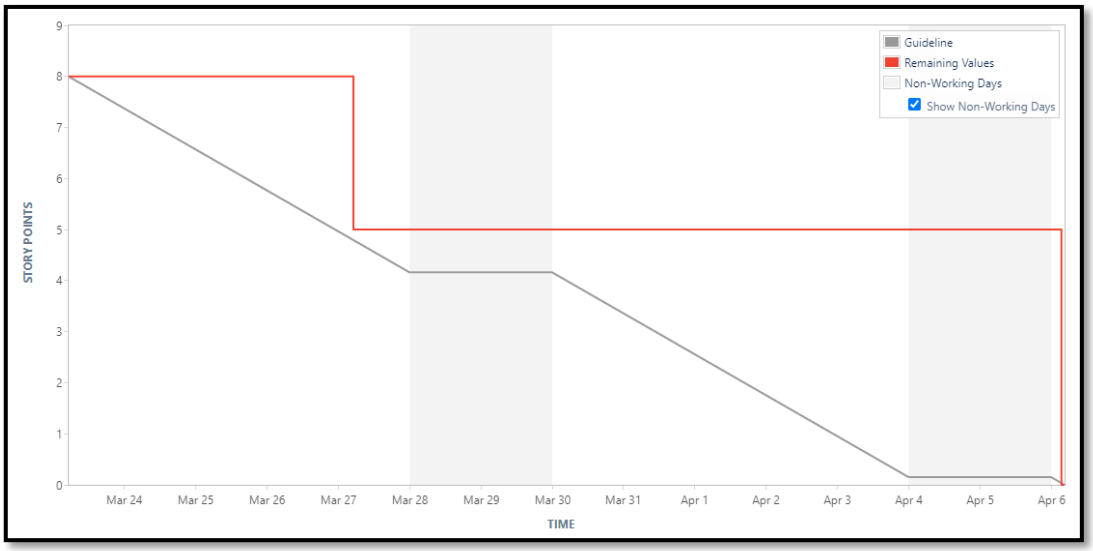

Figura 11. Burndown chart del sprint 3

La historia CCBLESDK-10 BLE communication se ha completado primero, sin ningún problema destacable.

CCBLESDK-8 Security Layer, por otra parte, no se ha podido completar. La implementación inicial está hecha, pero no funciona correctamente en las pruebas. La causa del retraso proviene del hecho de que fuera una historia más larga en general, sumado a una mayor dificultad para realizar pruebas trabajando desde casa.

La historia queda por tanto aplazada para el siguiente sprint.

2.4 Sprint 4 (6 abr. – 27 abr.)

En este sprint las tareas seleccionadas son las contenidas en la Tabla 15.

Historia	Resumen	Puntos de historia	Horas
CCBLESDK-8 (Parte II)	Security Layer	3	12
CCBLESDK-9	Prueba de componentes del API	5	20

Tabla 15. Tareas asignadas al sprint 4

Se continúa con el trabajo en la historia CCBLESDK-8, ya que no fue posible acabar a tiempo en el sprint anterior. Se han reducido a 2 los puntos de historia debido a que ya está escrito el código que da la funcionalidad y solo resta depurarlo.

2.4.1 CCBLESDK-8 – Security Layer (Parte II)

Para detectar cuál puede ser la raíz del problema, durante la reunión inicial del sprint he solicitado acceso al código de la aplicación usada para realizar las pruebas.

Las salidas producidas por dicha aplicación indican que el mal funcionamiento de la autenticación se debe a que, durante la comunicación, los dispositivos solo son capaces de enviarse datos de hasta 20 bytes, y cualquier dato de tamaño superior es cortado.

Las siguientes pruebas han sido ejecutadas para comprobar dónde puede estar originándose el error:

- Utilizar la aplicación Digi Mobile para abrir la conexión con un XBee corriente. En este caso, los datos se envían correctamente en ambas direcciones, y se puede superar la autenticación.
- Utilizar una aplicación general de escaneo de dispositivos BLE para conectarse al servidor GATT creado con el API en desarrollo, y enviar datos de más de 20 bytes. En este caso, también es posible leer y escribir correctamente los datos de mayor volumen.
- En el lado del cliente, pedir explícitamente una unidad máxima de transferencia (MTU) de datos superior a los 20 establecidos por defecto. La librería utilizada en la aplicación móvil tiene un método con esta función, pero al ejecutarlo se queda bloqueado, esperando a que ambos dispositivos acuerden un valor.

Además, el problema persiste al usar diferentes teléfonos móviles con Digi Mobile, así como al utilizar otra aplicación diferente (similar a Digi Mobile, desarrollada en el periodo de prácticas) que utiliza también la librería de C# Plugin.BLE.

La conclusión parece ser que hay algo que impide negociar un MTU diferente al dado por defecto en la comunicación entre la librería Plugin.BLE y el servidor GATT.

Las siguientes pruebas de interés se exponen a continuación:

- En Digi habían desarrollado anteriormente una aplicación Python que se conecta a un XBee sin interfaz BLE con el objetivo de utilizar la interfaz Bluetooth propia del sistema anfitrión que la ejecute y simular así una comunicación BLE real con otro dispositivo. La aplicación implementa una capa de seguridad similar a la de este API. Como se trata de un ejemplo ya probado en la empresa y que también fue utilizado durante el periodo de prácticas, debería funcionar sin problemas.
- Todas las pruebas hasta ahora se han realizado en móviles con sistema operativo Android. La aplicación Digi Mobile es multiplataforma y también tiene versión para iOS, así que sería interesante comprobar sin funciona correctamente en este caso.

Por desgracia, las dos pruebas requieren de dispositivos que fueron dejados en la oficina, y debido a la cuarentena no ha sido posible acceder a ellos fácilmente. Tras una espera de algunos días, uno de mis tutores por fin es autorizado para ir a recoger diversos materiales, y me trae todo lo necesario.

Procedo a realizar la primera prueba y como resultado se ha comprobado que tampoco es capaz de enviar ni recibir datos mayores a 20 bytes.

Por otra parte, la prueba en iOS muestra que los datos se transmiten correctamente entre los dos dispositivos, sin cortes como sucedía hasta ahora.

Al no poder determinar exactamente el origen del problema encontrado en Android, y no tener suficientes recursos para probarlo con más dispositivos diferentes, se ha tomado la decisión de dar prioridad al uso de iOS respecto a Android para no bloquear el desarrollo. La aplicación móvil a desarrollar será principalmente para iOS, y el soporte para Android se convierte en una posible mejora para el futuro.

Finalmente, se depuran algunos errores menores encontrados con la autenticación ya funcional y se cierra la historia.

2.4.2 CCBLESDK-9 – Prueba de componentes del API

Durante la espera para recibir el material necesario, es posible continuar el trabajo en otras tareas.

Una parte importante para poder comenzar a testear la funcionalidad desarrollada hasta ahora consiste en poder abrir conexión con la interfaz BLE del XBee. Esto es algo que no ha sido probado hasta ahora, por lo que podría no funcionar correctamente.

Los primeros intentos fallan, así que contacto a uno de mis tutores para que me ayude por videollamada. Tras la sesión, estos son los problemas que se han encontrado:

- Se estaba utilizando /dev/ttyXBee como nombre del puerto serie conectado al XBee, según aparecía en la documentación. El puerto utilizado realmente es /dev/ttymxc1.
- El XBee no estaba configurado correctamente, así que hubo que resetearlo.

Una vez resuelto todo, la comunicación entre el ConnectCore y el XBee ya es posible y podrán dar comienzo las pruebas de componentes definidas en el apartado 2.1.4 para comprobar el correcto funcionamiento del API.

Cabe destacar que hasta ahora la mayoría de las funciones del API sólo habían sido ejecutadas en el ordenador donde se lleva a cabo el desarrollo. Para simular el funcionamiento real, todas las pruebas de esta tarea serán ejecutadas en un ConnectCore, utilizando primero la interfaz BLE nativa y después la que proporciona el XBee conectado.

En la Tabla 16 se recuperan las pruebas definidas en la Tabla 9 del apartado 2.1.4 con los resultados de las pruebas.

Código	Prueba	Interfaz nativa	Interfaz XBee
PA-01	Comprobar con una aplicación estándar de escaneo de dispositivos BLE que el servidor anunciado es visible desde interfaz nativa o XBee, y que el cliente las ve de igual forma.	Superada	Superada
PA-02	Comprobar con una aplicación estándar de escaneo de dispositivos BLE que no sea posible establecer la comunicación sin haber completado el proceso de autenticación.	Superada	Superada
PA-03	Comprobar con una aplicación estándar de escaneo de dispositivos BLE que tras establecer conexión el cliente puede ver las características definidas independientemente de la interfaz BLE usada.	Superada	Superada
PA-04	Comprobar que al detener el servicio bluetooth el servidor ya no sea visible.	Superada	Superada
PA-05	Comprobar con una aplicación de escaneo de dispositivos BLE que implemente los protocolos de seguridad que los datos se envían correctamente.	Superada*	Superada*
PA-06	Comprobar con una aplicación de escaneo de dispositivos BLE que implemente los protocolos de seguridad que los datos se reciben correctamente.	Superada*	Superada*
PA-07	Comprobar que el API detecta cuándo hay un dispositivo conectado y cuándo no.	Superada	Superada
PA-08	Comprobar que tras cambiar el nombre del servicio anunciado una aplicación de escaneo de dispositivos BLE detecte el servidor con el nuevo nombre.	Superada	No superada

Tabla 16. Resultados de las pruebas del API

En las pruebas con PA-07 se encontraron algunos comportamientos diferentes a los esperados respecto a cuándo se reciben las señales de conexión y desconexión. En el caso de la desconexión, en ocasiones se recibe la señal varias veces. En el caso de la conexión, ninguna.

Además, se observó que la autenticación tomaba mucho tiempo, en algunos casos el suficiente para que fallara.

Los siguientes cambios se han implementado para resolver todos estos asuntos:

- En lugar de suscribir todos los objetos gestionados por el D-Bus, GATTServer ahora suscribe solo aquellos que detecta que son dispositivos que soportan BLE, lo cual evita la suscripción por parte de objetos que no tienen la propiedad “Connected” como pueden ser servicios o características, y reduce la carga soportada general.

- Al suscribir una nueva interfaz, controlar que dicha interfaz no esté ya escuchando la señal de conexión. Esto sucedía a veces si una interfaz era desconectada del D-Bus y volvía a conectarse. La forma de implementarlo ha sido almacenando todas las interfaces suscritas en una estructura de datos (inicialmente un vector).
- Cada vez que se desconecta una interfaz del D-Bus, darlo de baja de la señal a la que está escuchando y eliminarlo de la estructura de datos. Ya que para dar de baja era necesario un valor (representando la señal) obtenido al suscribir la interfaz, la estructura de datos cambia de un vector a un diccionario que almacena pares de forma {interfaz : señal}.

Las pruebas PA-05 y PA-06 se superaron solo parcialmente. Durante su ejecución, solo funcionó la comunicación entre Digi Mobile y la interfaz XBee, y entre la aplicación desarrollada en prácticas y la interfaz nativa. Sin embargo, esto parece ser un problema de las aplicaciones móviles, ya que Digi Mobile está enfocada a la comunicación con un XBee (y no con un ConnectCore), y la otra aplicación tiene desactualizadas algunas de sus librerías. Por ello se determinará si es un problema en las pruebas globales del SDK y no ahora.

Finalmente, hay un error la prueba PA-08 que impide darla por superada. Mientras que en la interfaz nativa funciona como estaba previsto, no sucede así en la interfaz XBee. Al realizar la petición para cambiar el parámetro del XBee que se encarga del nombre anunciado, salta una excepción. Tras una reunión con los tutores se ha determinado que el problema surge de que el XBee utilizado para las pruebas tiene una versión de firmware antigua, que no posee dicho parámetro. No ha sido posible probarlo con una versión más reciente del firmware, pero como la funcionalidad testeada proviene de una simple llamada a un método de la clase XBeeDevice, se da por hecho que funcionará correctamente en un entorno real. Si no fuera así, corregirlo sería sencillo por lo simple que es su implementación.

Tras esto el API ya cumple con los criterios de aceptación necesarios.

2.4.3 Planificación de aplicación Python

Debido a la situación en la que se encuentra el sprint (distribuido en 3 semanas en lugar de las típicas 2), y ya que se ha perdido algo de tiempo sin poder continuar el trabajo, se extiende un poco la duración para poder incorporar alguna nueva tarea aprovechando que quedan algunos días libres.

Para ello, primero se lleva a cabo la estimación de las historias y tareas del nuevo bloque de desarrollo.

Código	Resumen	Tareas	Descripción	Puntos de historia
CCBLESDK-24	Análisis y diseño de la aplicación	TP-01, TP-02	Requisitos, casos de uso, diseño de clases y de casos de prueba.	3
CCBLESDK-26	Implementar aplicación	TP-03, TP-04	Como cliente, quiero tener una aplicación de ejemplo que demuestre cómo usar el API ConnectCore BLE, para poder desarrollar mis propias aplicaciones basadas en ese ejemplo.	5

Tabla 17. Tareas e historias correspondientes a la aplicación Python

La anterior tarea de análisis y diseño fue estimada con 5 puntos de historia, y aun así se alargó más de lo planificado. Aun teniendo eso en cuenta, esta vez se ha estimado con 3; es una aplicación muy pequeña y está basada en el API ya creado, así que es difícil que su diseño se complique.

Es notable la ausencia de una tarea de prueba de componentes en este bloque. Se ha decidido que, por depender la aplicación Python de la aplicación móvil que se va a desarrollar en sprints posteriores, se juntarán las dos tareas de prueba en una sola una vez acabado todo lo demás.

Tras la estimación, da comienzo el trabajo en la primera tarea, el análisis y el diseño de la aplicación Python.

2.4.4 Análisis de la aplicación Python

Con esta tarea comienza el trabajo en el siguiente bloque de desarrollo del SDK; una aplicación Python que sirva como ejemplo de uso del API.

En términos generales, va a consistir en una aplicación simple que utilice todas las funciones que proporciona el API para ofrecer un servicio que permita cambiar o acceder a las propiedades de unos falsos ficheros de configuración de interfaces de red (dos en concreto, “Ethernet” y “WiFi”), según las peticiones que envíe el cliente a través de BLE.

2.4.4.1 Identificación de los usuarios de la aplicación Python

En el uso de la aplicación Python, se ha detectado como principal rol que asuma el usuario el de “cliente”.

Este usuario se trata de un desarrollador que utiliza esta aplicación Python a modo de ejemplo para desarrollar sus propias aplicaciones para la comunicación con un ConnectCore.

2.4.4.2 Catálogo de requisitos la aplicación Python

Los requisitos funcionales que debe satisfacer la aplicación se muestran en la Tabla 18.

Código	Requisito
R-01	La aplicación hará uso de todas las funciones del API.
R-02	La aplicación utilizará el API para crear un servidor GATT con dos características, a partir de las cuales se enviarán y recibirán datos.
R-03	La aplicación podrá leer y escribir datos en un fichero para modificar una falsa configuración de interfaces de red (“Ethernet” y “WiFi”).
R-04	La aplicación definirá su propio protocolo de comunicación con el cliente, que será utilizado dentro de los paquetes del protocolo que ya define el API.
R-05	La aplicación permitirá customizar el nombre del servicio anunciado.

Tabla 18. Requisitos funcionales de la aplicación Python

El cumplimiento de todos estos requisitos tiene prioridad máxima, pues se refieren a la funcionalidad básica que debe tener la aplicación para poder ser un buen ejemplo.

Por otra parte, también tiene una serie de requisitos no funcionales que debe satisfacer igualmente. Por estar haciendo uso de todas las funciones del API, los requisitos no funcionales que se definieron para este se aplican también aquí. Como requisito extra se incluye:

- **Requisitos de documentación**: el código deberá estar bien documentado, explicando en detalle lo que está ocurriendo por cada bloque funcional.

2.4.4.3 Casos de uso de la aplicación Python

En la Figura 12 se expone el diagrama de casos de uso de la aplicación. Se considera el rol de cliente mencionado en la sección 2.5.3.1 como el único actor.

Esta vez nos encontramos ante un diagrama de casos de uso mucho más sencillo que el del API, ya que se trata de una aplicación de ejemplo enfocada a mostrar las funcionalidades del API de la manera más simple posible.

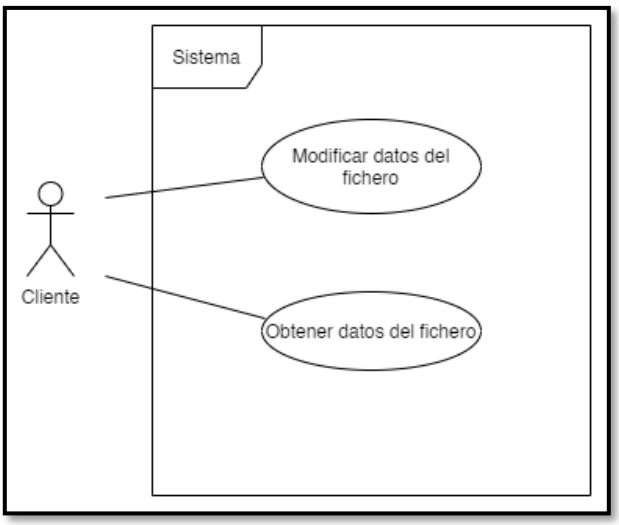

Figura 12. Diagrama de casos de uso de la aplicación Python

2.4.5 Diseño de la aplicación

2.4.5.1 Diseño del protocolo de comunicación

En la comunicación entre el servidor GATT creado con el API y el dispositivo cliente, conviene implementar un protocolo que gestione las peticiones y respuestas generadas de una forma más eficiente. En esta aplicación se definirá un protocolo simple que sirva para que los usuarios del API puedan tener una referencia a la hora de definir el suyo propio.

El protocolo estará construido sobre JSON, por ser un formato en texto y fácil de entender, con librerías que facilitan su uso.

Lo mostrado en la Figura 13 es la estructura que van a seguir los mensajes.

```
{
  "Operation" : "Read" / "Write",
  "Interface" : "Ethernet" / "WiFi",
  "Type" : "Static" / "Dynamic",
  "IP Address" : "xxx.xxx.xxx.xxx",
  "Default Gateway" : "xxx.xxx.xxx.xxx",
  "Subnet Mask" : "xxx.xxx.xxx.xxx",
  "DNS Server 1" : "xxx.xxx.xxx.xxx",
  "DNS Server 2" : "xxx.xxx.xxx.xxx",
  "MAC Address" : "xxx.xxx.xxx.xxx",
  "Enabled" : "True" / "False",
  "Status" : "Error" / "OK"
  }
```

Figura 13. Estructura del protocolo JSON

En la Tabla 19 se muestran todos los posibles valores que admite el formato JSON especificado. Cualquier otro caso será considerado como no válido.

Operación	Propiedades	Requerido	Valor
Write	Interface	Sí	Ethernet / WiFi
	Type	Sí	Static / Dynamic
	IP Address	Si “Type” es “Static”	xxx.xxx.xxx.xxx
	Default Gateway	Si “Type” es “Static”	xxx.xxx.xxx.xxx
	Subnet Mask	Si “Type” es “Static”	xxx.xxx.xxx.xxx
	DNS Server 1	No	xxx.xxx.xxx.xxx
	DNS Server 2	No	xxx.xxx.xxx.xxx
	MAC Address	No	xx:xx:xx:xx:xx:xx
	Enabled	No	True / False
Read	Interface	Sí	Ethernet / WiFi

Tabla 19. Posibles valores para los campos del JSON

Respecto al campo status, si el mensaje es una petición del cliente, estará vacío; si el mensaje es correcto, la aplicación lo devolverá con “OK” en el campo; y si el mensaje ha producido un error, lo devolverá con “Error” seguido de una pequeña explicación.

2.4.5.2 Diseño de clases

El diagrama de la Figura 14 muestra el diseño de las clases que seguirá la implementación de la aplicación Python de ejemplo. Está compuesta por una sola clase que utiliza todas las funciones disponibles en BLEService, de manera que sea un ejemplo completo y fácil de entender.

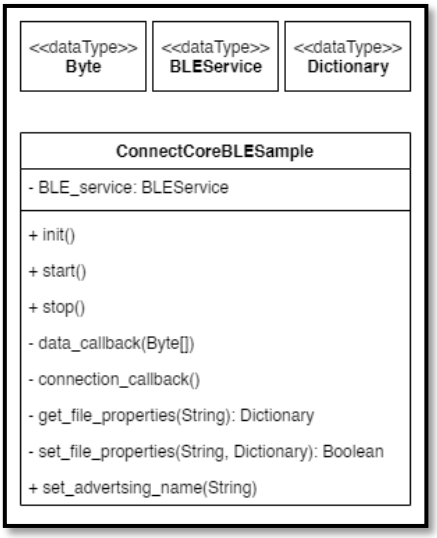

Figura 14. Diagrama de clases de la aplicación

A continuación, se explican brevemente los métodos de ConnectCoreBLESample.

init: mediante get_instance de BLEService, genera una instancia de esta clase y la almacena en el atributo **BLE_service**. Además, pasa al servicio dos funciones a las que llamar en distintos eventos:

- data_callback se añade como función de callback para cuando el servicio recibe nuevos datos con add_data_received_callback.
- connection_callback se añade como función de callback para cuando el servicio recibe una nueva conexión con add_connect_callback.

start: comienza el servicio Bluetooth de BLE_service mediante el método start_service.

stop: detiene el servicio Bluetooth de BLE_service mediante el método stop_service.

data_callback: la función a llamar cuando se reciben nuevos datos. Esta función recibe datos en formato JSON, y los trata para ejecutar distintas respuestas:

- Si hay errores al tratar la cadena JSON, envía de vuelta un mensaje de error.
- Si en la cadena JSON se especifica que se quiere hacer una operación de escritura, accede a un fichero y modifica las propiedades que se quieran cambiar con los valores especificados en los datos. Tras ello, envía un mensaje de confirmación si todo ha ido bien.
- Si en la cadena JSON se especifica que se quieren leer propiedades, se accede a un fichero para obtener su valor, y tras ello se envía de vuelta.

connection_callback: la función a llamar cuando establece una nueva conexión. Cuando es llamada, muestra un mensaje por pantalla que expone la conexión y además llama a get_type de BLE_service para indicar la interfaz que está utilizando el ConnectCore para conectarse.

get_file_property: accede a un fichero de configuración especificado en el parámetro, que corresponde a una interfaz falsa de internet. Obtiene el valor de todas las propiedades que contiene y las devuelve dentro de un diccionario.

set_file_property: accede un fichero de configuración especificado en el primer parámetro que corresponde a una interfaz falsa de internet. Modifica todas las propiedades para cambiarlas según los valores del diccionario del segundo parámetro.

set_advertising_name: cambia el nombre del servicio de BLE_service anunciado al pasado en el parámetro.

2.4.5 Diseño de pruebas de la aplicación

En esta sección se definen las pruebas a las que se someterá la aplicación de ejemplo una vez terminada.

Como parte de la última tarea de la componente, se incluirá una sección en la que se indicará una por una si ha sido posible superar la prueba o no.

Código	Prueba
PP-01	Comprobar que la aplicación se conecta correctamente con un cliente.
PP-02	Comprobar que la aplicación muestra correctamente la interfaz Bluetooth utilizada.
PP-03	Comprobar que la aplicación obtiene correctamente los datos de los falsos ficheros de configuración y que el cliente los recibe correctamente tras una petición.
PP-04	Comprobar que la aplicación modifica correctamente los datos de los falsos ficheros de configuración cuando el cliente hace una petición, y que este recibe un mensaje de confirmación.
PP-05	Comprobar que, tras cambiar el nombre anunciado, este se muestra correctamente.

Tabla 20. Diseño de pruebas de la aplicación Python

2.4.6 Revisión del sprint

La información sobre las horas dedicadas a cada tarea se recoge en la Tabla 21.

Tarea	Horas estimadas	Horas reales	Desviación
Security Layer	12	30	+66,7 %
Prueba de componentes del API	20	6	-70 %
Análisis y diseño de la aplicación	12	6	-50 %
Documentación	8	4	-50 %

Tabla 21. Comparativa de horas dedicadas en el sprint 4

La Figura 15 muestra la evolución del sprint, y en ella se puede ver un resumen visual del tiempo que se ha tardado en completar las historias.

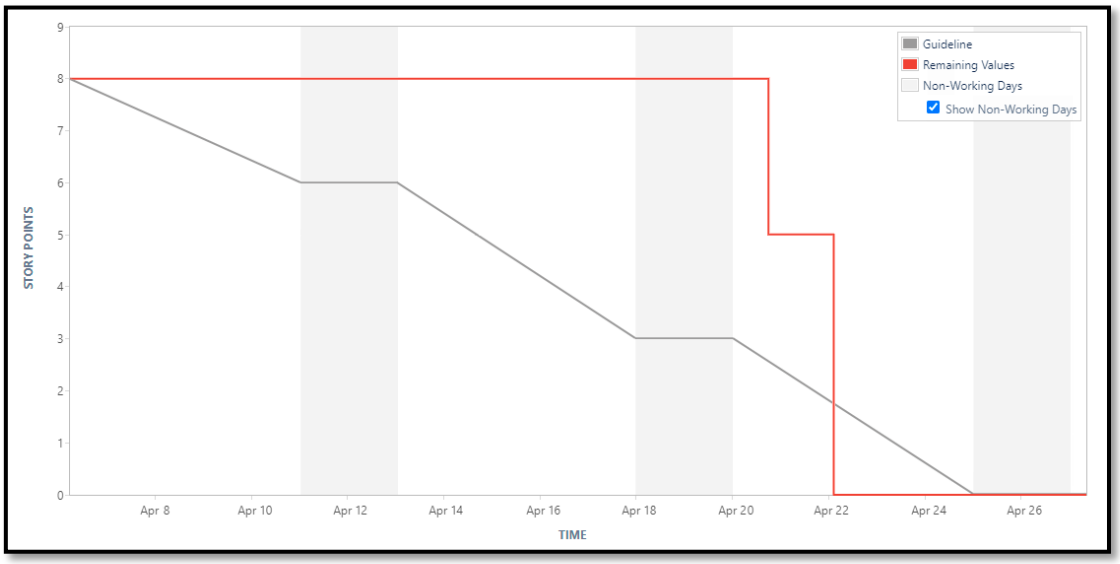

Figura 15. Burndown chart del sprint 4

En el diagrama se aprecia que las primeras semanas no han sido demasiado productivas, principalmente a causa de los contratiempos encontrados y por no disponer del material requerido para hacerles frente. Sin embargo, vez conseguido el material necesario, el desarrollo ha avanzado mucho más rápidamente.

Si miramos al gráfico podemos ver que todas las historias y tareas han sido completadas en la última semana a pesar de algunos de los problemas encontrados durante los tests.

Por otra parte, aunque no se muestre, también se ha abordado una tarea de análisis y diseño correspondiente a la aplicación Python que no estaba inicialmente planificada para este sprint. Al tratarse de una aplicación más sencilla con pocas clases y métodos, el tiempo necesario para esta tarea ha sido algo menor de lo esperado, así que se ha podido completar en su totalidad.

A causa de esto todo, a este sprint se le ha dedicado algo más de las 40 horas planificadas inicialmente, aunque se ha considerado oportuno por todo el tiempo perdido en problemas que no podían haber sido previstos, y porque el sprint se encontraba situado en lapso de tiempo de 3 semanas.

2.5 Sprint 5 (27 abr. – 11 may.)

En este sprint, además de completar la tarea restante de la aplicación Python, da comienzo el trabajo en la aplicación móvil que va a formar parte del ejemplo de uso del API. Por ello, conviene comenzar estimando las historias y tareas que formarán parte de este último bloque de desarrollo.

Código	Resumen	Tareas	Descripción	Puntos de historia
CCBLESDK-33	Análisis y diseño de la aplicación móvil	TM-01, TM-02	Requisitos, casos de uso, diseño de clases y de casos de prueba.	3
CCBLESDK-34	Implementar aplicación móvil	TM-03, TM-04	Como cliente, quiero tener un ejemplo de aplicación móvil que se conecte con un dispositivo ConnectCore 6UL ejecutando una demo de configuración de redes, para poder ver el API (corriendo en el ConnectCore) en acción.	5
CCBLESDK-25	Prueba de componentes de la aplicación móvil	TM-05	Ejecutar los casos de prueba definidos en la fase de diseño.	3

Tabla 22. Tareas e historias correspondientes a la aplicación Python

Las tareas e historias seleccionadas para el sprint son las mostradas en la tabla 23.

Historia	Resumen	Puntos de historia	Horas
CCBLESDK-26	Implementar aplicación	5	20
CCBLESDK-33	Análisis y diseño de la aplicación móvil	3	12

Tabla 23. Tareas asignadas al sprint 5

La aplicación Python, como ya se ha mostrado en el análisis y en el diseño, es bastante sencilla, por lo que basta con 5 puntos de historia para terminarla.

Respecto al análisis y diseño de la aplicación móvil, aunque requiera de apartados que no estaban presentes en la aplicación Python como el diseño de la interfaz gráfica, mantiene una estimación de 3 puntos de historia debido a que el protocolo de comunicación ya está definido y a que comparte muchos de sus requisitos.

2.5.1 CCBLESDK-26 – Implementar aplicación

Como se definió en la etapa de diseño, la implementación consiste en una única clase, **ConnectCoreBLESample**, con una serie de métodos que utilizan los métodos del API (concretamente de la clase BLEService) para mostrar un ejemplo de cómo utilizarlos.

start y **stop** abren y cierran el servicio respectivamente, mediante los métodos start_service y stop_service respectivamente.

_data_received_callback es la función a llamar cuando se reciben datos (una cadena de bytes que contiene una cadena JSON codificada) en el servicio, y realiza varias funciones:

- Convierte la cadena JSON codificada que ha sido recibida en un diccionario mediante la librería json.

- Comprueba la validez de la cadena JSON recibida. Para realizar la validación, se utiliza el método **_check_json**[7], que verifica si los datos recibidos siguen el formato definido en la etapa de análisis. En caso de que no sea así, devuelve una excepción indicando el error encontrado.
- Si los datos recibidos indican que se trata de una operación de escritura, utiliza el método **_set_file_properties** para sobrescribir los datos del fichero de propiedades especificado. Este método obtiene las propiedades que estaban ya presentes en el fichero y las actualiza con los nuevos valores.
- Si los datos recibidos indican que la operación es de lectura, utiliza el método **_get_file_properties** para leer todas propiedades contenidas en el fichero especificado, las convierte en una cadena JSON y las envía con send_data.
    - En ambas operaciones, si la operación ha sido completada sin problemas, se envía de vuelta un mensaje de confirmación indicando la operación y el estado ("Status": "OK") con send_data. En el caso de la lectura este mensaje de confirmación se incluye entre las propiedades enviadas.
    - Si hubiera habido alguna excepción, se enviaría en su lugar un mensaje indicando la operación y la causa del error ("Status": "Error: …").

_connection_callback es una función a la que se llama cada vez que se entabla conexión con un cliente. Servirá para mostrar por pantalla una cadena en la que se indique qué interfaz se está utilizando para conectarse, utilizando el método get_type.

change_advertising_name cambia el nombre que se mostrará a otros dispositivos BLE cuando se anuncie. Utiliza configure_advertising_name.

Por último, también se ha implementado un método main en el que comenzará la ejecución del servicio. Este método crea un objeto de tipo ConnectCoreBLESample, e inicia un bucle en el que permite al usuario realizar diversas opciones según la cadena de entrada que initroduzca:

- start: inicia el servicio si no está activo y empieza a anunciarlo.
- stop: detiene el servicio si está activo y deja de anunciarlo.
- name: pide al usuario introducir un nuevo nombre anunciado.
- password: pide al usuario introducir una nueva contraseña para la autenticación, utilizando el método set_password de BLEService, y la librería getpass para que no se muestre mientras se escribe.
- exit: finaliza la ejecución del programa, y detiene el servicio si estaba activo.

[7] Inicialmente se valoró la opción de realizar esa comprobación mediante un JSON schema, que se trata de un esquema también hecho en JSON que permite validar datos JSON según el formato que define. Existen librerías que facilitan su uso tanto en Python como en C#, y además permitiría utilizar el mismo esquema tanto para validar en servidor como en cliente.

Sin embargo, se rechazó su uso por varias razones. Primero, porque hace más difícil mostrar los errores encontrados, ya que su método de validación devuelve errores genéricos siempre que encuentra un problema (es posible crear mensajes de error personalizados, pero la documentación no deja muy claro cómo). Segundo y más importante, porque al tratarse de una aplicación de ejemplo, debería ser lo más simple posible, y utilizar JSON schema significaría introducir una tecnología nueva que podría confundir al usuario.

Finalmente se optó en su lugar por un método que devuelve una excepción al encontrarse con un problema, de igual forma que haría el método de validación de la librería jsonschema.

2.5.2 Análisis de la aplicación móvil

Aquí da comienzo el trabajo en el último bloque de desarrollo del SDK; una aplicación móvil que se conecte a la aplicación Python ya desarrollada, de tal forma que permita que ésta utilice todos sus métodos y, en consecuencia, los métodos del API.

La aplicación en sí será una aplicación que rastrea dispositivos ConnectCore 6UL cercanos mediante BLE, y que permite conectarse a ellos tras autenticarse. Una vez entablada la conexión, se comunicará con la aplicación Python desplegada en el ConnectCore para obtener datos de falsos ficheros de configuración de redes ("Ethernet" y "WiFi") y mostrárselos al usuario, y también le permitirá modificarlos.

2.5.2.1 Identificación de los usuarios de la aplicación móvil

En el uso de la aplicación móvil, se ha detectado como principal rol que asuma el usuario el de "cliente".

Este usuario se trata de un desarrollador que utiliza esta aplicación móvil a modo de ejemplo para desarrollar sus propias aplicaciones para la comunicación con un ConnectCore que corra una aplicación desarrollada con el API.

2.5.2.2 Catálogo de requisitos la aplicación móvil

Los requisitos funcionales que debe satisfacer la aplicación se muestran en la Tabla 24.

Código	Requisito
R-01	La aplicación permitirá hacer uso de todos los métodos de la aplicación Python en el ConnectCore conectado.
R-02	La aplicación permitirá rastrear dispositivos ConnectCore 6UL mediante BLE.
R-03	La aplicación permitirá conectarse con uno de esos dispositivos, y permitirá autenticarse frente a él.
R-04	La aplicación permitirá elegir entre las interfaces de red "Ethernet" y "WiFi" para mostrar sus datos y configurarlas.
R-05	La aplicación permitirá leer todos los parámetros de la interfaz escogida.
R-06	La aplicación permitirá sobrescribir todos los parámetros de la interfaz escogida.
R-07	La aplicación utilizará el mismo protocolo de comunicación que el ya definido para la aplicación Python.

Tabla 24. Requisitos funcionales de la aplicación móvil

El cumplimiento de todos estos requisitos tiene prioridad máxima, pues se refieren a la funcionalidad básica que debe tener la aplicación para poder ser un buen ejemplo.

Por otra parte, también tiene una serie de requisitos no funcionales que debe satisfacer igualmente. Se consideran para esta aplicación los mismos requisitos no funcionales que se definieron para la aplicación Python, incluyendo los **requisitos de documentación**.

2.5.2.3 Casos de uso de la aplicación móvil

En la Figura 16 se expone el diagrama de casos de uso de la aplicación. Se considera el rol de cliente mencionado en la sección 2.5.3.1 como el único actor.

Esta vez nos encontramos ante un diagrama de casos de uso mucho más sencillo que el del API, ya que se trata de una aplicación de ejemplo enfocada a mostrar las funcionalidades del API de la manera más simple posible.

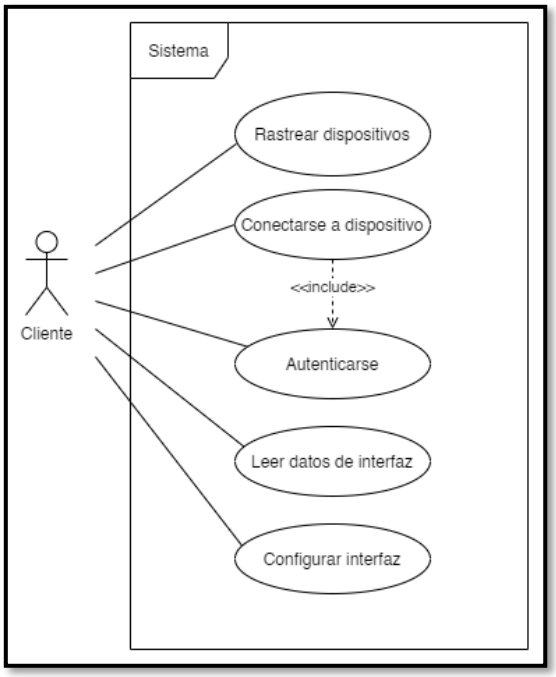

Figura 16. Diagrama de casos de uso de la aplicación móvil

2.5.3 Diseño de la aplicación

Para el diseño de la aplicación móvil, se seguirá un proceso ligeramente diferente al de las otras dos componentes del SDK.

En primer lugar, por incorporar una interfaz gráfica de usuario, se hace necesario elaborar un prototipo de la aplicación que dirija el desarrollo del front end.

En segundo lugar, el diseño de clases va a ser algo distinto, ya que en el desarrollo de una aplicación móvil con Xamarin.Forms está fuertemente recomendado utilizar el patrón modelo-vista-modelo de vista (MVVM). El patrón MVVM tiene ciertas similitudes con el patrón modelo-vista-controlador (MVC) visto durante la carrera, y de hecho comparte dos de sus componentes:

- El modelo, representando capa de datos y la lógica de negocio de la aplicación.
- La vista, encargada de mostrar al usuario los datos del modelo y la interfaz gráfica a través de la cual se comunicará con el resto del sistema.

La principal diferencia entre MVC y MVVM radica en el último componente, y es esta diferencia la que hace que MVVM sea preferible en una aplicación Xamarin:

- El controlador de MVC se encarga de responder a peticiones del usuario, modificando y accediendo a los datos del modelo, y de enviarlos a la vista para mostrarlos si se solicita un cambio en la forma en que se presentan.
- El modelo de vista de MVVM actúa como la lógica de negocio de la vista. Se encarga de la comunicación entre vista y modelo, de forma que el intercambio de datos se lleva a cabo de forma automática, a partir de un enlazador que sincroniza los datos entre las distintas capas cada vez que tiene lugar un cambio en cualquiera de ellas.

Como los cambios en el modelo sin peticiones del usuario van a ser frecuentes (por ejemplo, en el descubrimiento de nuevos dispositivos BLE), los enlaces de datos del modelo de vista permiten actualizar las vistas automáticamente en cuanto tengan lugar.

2.5.3.1 Diseño de la interfaz

La Figura 17 muestra un prototipo a bajo nivel de las pantallas con las que contará la aplicación final, y la interacción entre ellas.

Se trata pues de una primera pantalla inicial en la que se listan los distintos dispositivos encontrados durante el proceso de descubrimiento. Al pulsar en uno, la aplicación se dirige a una pantalla en la que se muestran algunos datos del dispositivo, entre ellos las interfaces que tiene listas para configurar. Al pulsar en una de ellas, la aplicación se dirige a la pantalla de configuración, en la que se pueden leer y modificar los parámetros ilustrados.

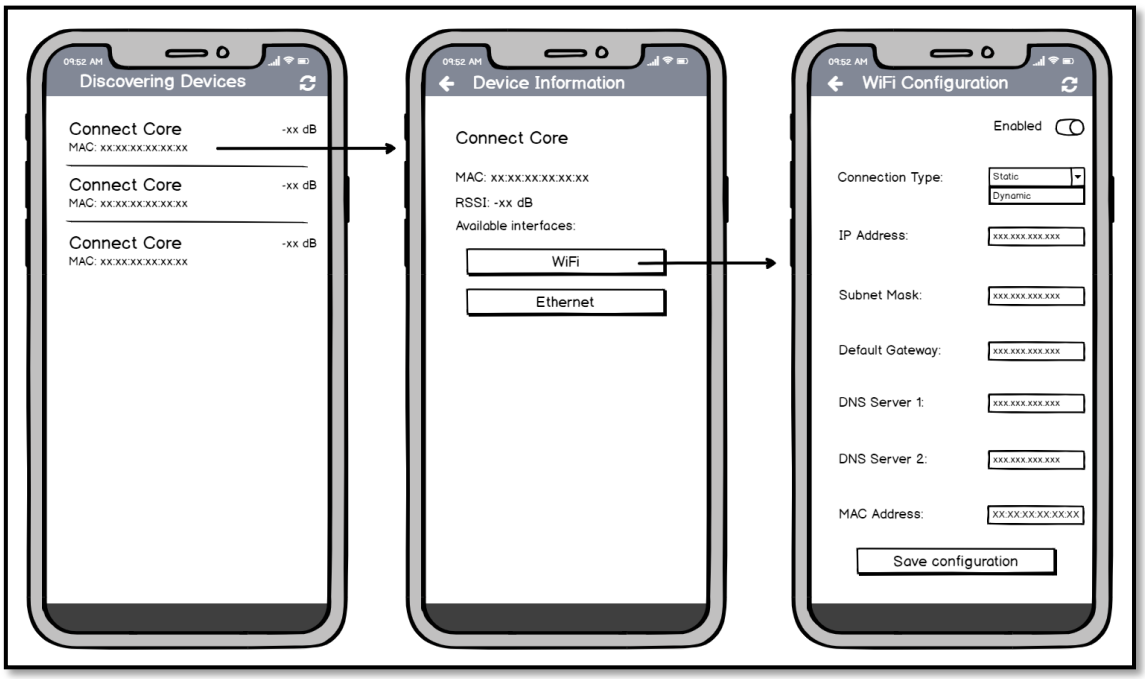

Figura 17. Prototipo a bajo nivel de la aplicación móvil

2.5.4.2 Diseño de clases

El diagrama de la Figura 18 muestra el diseño de las clases que seguirá la implementación de la aplicación móvil según el patrón MVVM expuesto anteriormente.

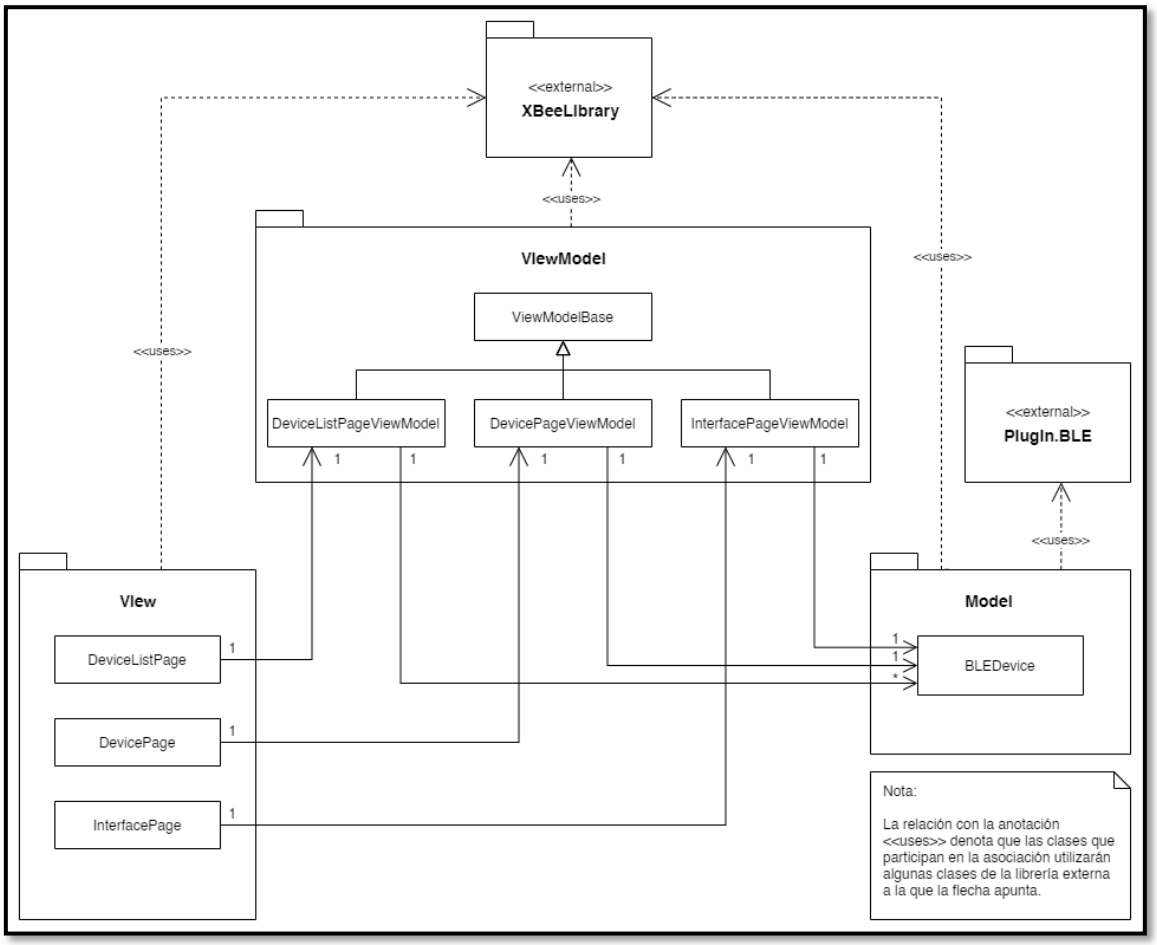

Figura 18. Diagrama de paquetes representando la aplicación

Cada uno de los paquetes del diagrama corresponde con una componente en MVVM.

El paquete View contiene las clases que representan las distintas páginas de la aplicación. Como estamos trabajando con Xamarin.Forms, además contiene ficheros XAML asociados a cada clase para definir el aspecto gráfico de la página:

- **DeviceListPage**, la página de inicio en la que se mostrarán los resultados de la búsqueda de dispositivos BLE, y se podrá seleccionar uno de ellos para acceder a sus datos. Corresponde con la primera ventana de la Figura 17.
- **DevicePage**, la página en la que se muestra la información del dispositivo escogido en DeviceListPage, y en la que se puede seleccionar una interfaz para configurar. Corresponde con la segunda ventana de la Figura 17.
- **InterfacePage**, la página en la que se exponen todos los parámetros de la interfaz seleccionada en DevicePage, y en la que se permite al usuario modificarlos. Corresponde con la última ventana de la Figura 17.

El paquete ViewModel contiene clases que controlan los datos que se muestran en las clases del paquete View:

- **ViewModelBase**, la clase base para las demás, y contiene métodos básicos para el enlace de datos con las vistas.
- **DeviceListPageViewModel** gestiona la búsqueda de dispositivos BLE y la conexión con ellos.
- **DevicePageViewModel** gestiona la comunicación con el dispositivo BLE seleccionado para obtener información relevante.
- **InterfacePageViewModel** gestiona la comunicación con el servidor GATT que implementa el servicio de configuración de interfaces.

El paquete Model contiene las clases que representan la capa de lógica de negocio:

- BLEDevice representa a un dispositivo BLE, con sus propiedades y métodos relevantes (indicador de fuerza de la señal recibida, dirección MAC etc.).

Por último, XBeeLibrary y Plugin.BLE son dos librerías externas que se utilizarán en toda la aplicación para gestionar toda la comunicación con XBees o dispositivos equivalentes (sirve para un servidor GATT con los mismos IDs para sus servicios y características) y para la comunicación por medio de BLE en general.

2.5.5 Diseño de pruebas de la aplicación

En esta sección se definen las pruebas a las que se someterá la aplicación de ejemplo una vez terminada.

Como parte de la última tarea de la componente, se incluirá una sección en la se indicará una por una si ha sido posible superar la prueba o no.

Código	Prueba
PA-01	Comprobar que la aplicación rastrea correctamente dispositivos BLE anunciándose.
PA-02	Comprobar que la aplicación se conecta correctamente con un servidor.
PA-03	Comprobar que la aplicación detecta correctamente las interfaces de red que permite configurar el servicio.
PA-04	Comprobar que la aplicación lee los datos de una interfaz implementada en el servidor.
PA-05	Comprobar que los datos se actualizan correctamente tras modificar la configuración de una interfaz.

Tabla 25. Diseño de pruebas de la aplicación móvil

Junto con estas pruebas se comprobarán las ya definidas en el pasado para la aplicación Python.

2.5.6 Revisión del sprint

La información sobre las horas dedicadas a cada tarea se recoge en la Tabla 26.

Tarea	Horas estimadas	Horas reales	Desviación
Implementar aplicación	20	30	+50 %
Análisis y diseño de la aplicación móvil	12	8	-33,3 %
Documentación	8	4	-50 %

Tabla 26. Comparativa de horas dedicadas en el sprint 5

La Figura 19 muestra la evolución del sprint, y en ella se puede ver un resumen visual del tiempo que se ha tardado en completar las historias.

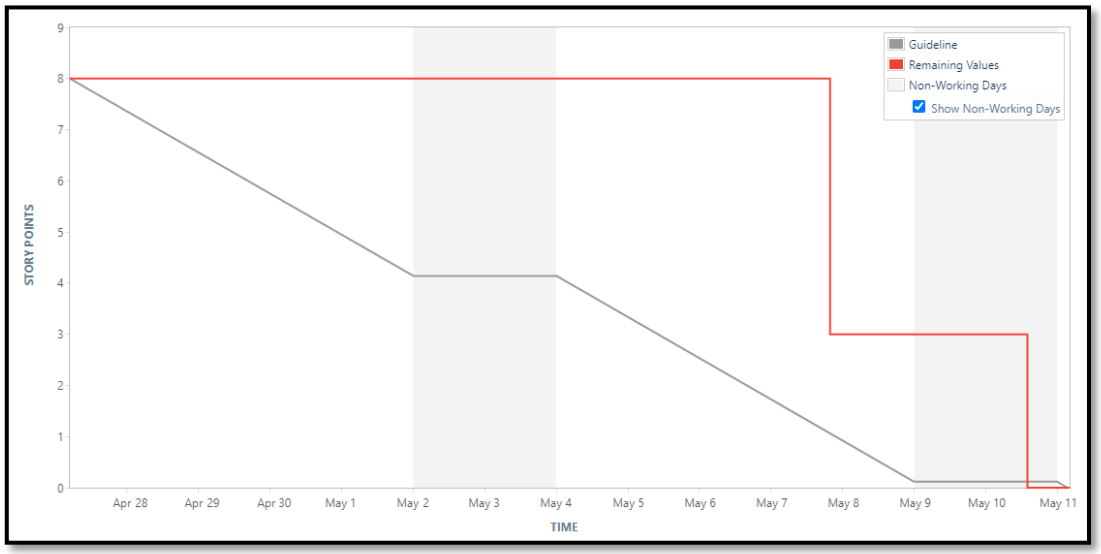

Figura 19. Burndown chart del sprint 5

La mayor parte del sprint ha estado dedicada a la aplicación Python. Esta historia se ha extendido algo más de lo esperado por tratarse de la implementación de un nuevo bloque de desarrollo completo, de modo que en el siguiente sprint se tendrá en cuenta que la implementación de la aplicación móvil podría necesitar de algo más de tiempo.

De todas formas, la desviación respecto a lo planificado no es tan severa como aparenta el gráfico, pues en la segunda semana se ha estado trabajando al mismo tiempo en las dos historias desde el miércoles 6, para alejar un poco la monotonía de la programación y resolución de errores.

2.6 Sprint 6 (11 abr. – 29 may.)

El sexto es el último sprint que va a formar parte del desarrollo del proyecto. Inicialmente se planificó que su duración se extendiera hasta el 25 de mayo, pero se van a utilizar los 4 días de holgura mencionados en la planificación por si el tiempo no fuera suficiente, dado que aún

queda implementar toda la aplicación móvil y realizar las pruebas generales. Aun así, se mantiene que la duración del sprint equivale a 8 puntos de historia.

En la Tabla 27 se muestran las historias y tareas por realizar.

Historia	Resumen	Puntos de historia	Horas
CCBLESDK-34	Implementar aplicación móvil	5	20
CCBLESDK-25	Prueba de componentes de la aplicación móvil	3	12

Tabla 27. Tareas asignadas al sprint 6

2.6.1 CCBLESDK-34 – Implementar aplicación móvil

Para esta historia será necesario implementar todas las componentes definidas en diseño según dicta el patrón MVVM (el modelo, las vistas y los modelos de vista).

Con el objetivo de acelerar el trabajo, se va a reutilizar parte del código utilizado en otra aplicación ya usada antes en el desarrollo y que también sigue el patrón MVVM: Digi XBee Mobile. Esta parte corresponderá a todo el apartado del descubrimiento de dispositivos BLE, que se incluirá en la página DeviceListPage y su modelo de vista DeviceListPageViewModel. La reutilización de la capa visual también servirá para darle un aspecto más homogéneo a la aplicación, en línea con otros productos de la empresa.

También se van a utilizar las librerías XBeeLibrary.Core y XBeeLibrary.Xamarin de Digi, que poseen clases y métodos utilizados para la comunicación por BLE con un XBee. Aunque es posible que el dispositivo conectado sea un ConnectCore a través de su interfaz BLE nativa, al utilizar servicios y características iguales a los de un XBee la comunicación con él funciona de forma similar.

En primer lugar, respecto al modelo de la aplicación, está compuesto únicamente por la clase **BLEDevice**, que representa un dispositivo con interfaz BLE y asociado a un XBee. Esta clase cuenta con los siguientes métodos, atributos y propiedades de interés:

- Hereda de la interfaz INotifyPropertyChanged el evento PropertyChanged, de forma que puede notificar si alguna de sus propiedades cambia su valor.
- Propiedad **Device** de la clase IDevice, de la librería Plugin.BLE. El objeto que gestiona la conexión y comunicación BLE. No es utilizado directamente por la clase, sino que se le pasa a XBeeDevice al crearlo y este lo gestiona por su cuenta. Los métodos **Open** y **Close** se utilizan para abrir y cerrar su interfaz BLE (incluyendo todo lo relacionado con la autenticación).
- Propiedad **XBeeDevice** de la clase XBeeBLEDevice, de la librería XBeeLibrary.Xamarin. Representa el dispositivo XBee asociado al objeto, y gestiona toda la interfaz BLE.

Respecto a las vistas y a sus modelos de vista, los podemos separar según las páginas a las que pertenecen.

Antes de eso, vamos a ver algunos de los aspectos que tienen todos en común.

- Todas las vistas heredan de CustomContentPage, para poder implementar un botón de regreso a la página anterior diferente al que está presente por defecto (por ejemplo, para que muestre un mensaje de aviso al pulsarlo).
- Todas las vistas guardan el modelo de vista al que están asociadas en una propiedad.
- Todos los modelos de vista heredan de la clase ViewModelBase, que a su vez hereda de la interfaz INotifyPropertyChanged. Esta clase ofrece algunas funcionalidades básicas que van a compartir todos los modelos de vista, como métodos para mostrar alertas del sistema en forma de pop-ups, para obtener la página en la que se encuentra el usuario en un determinado momento, o para gestionar pop-ups con diálogos de carga.

En la Figura 20 se muestra el aspecto final de la pantalla de inicio.

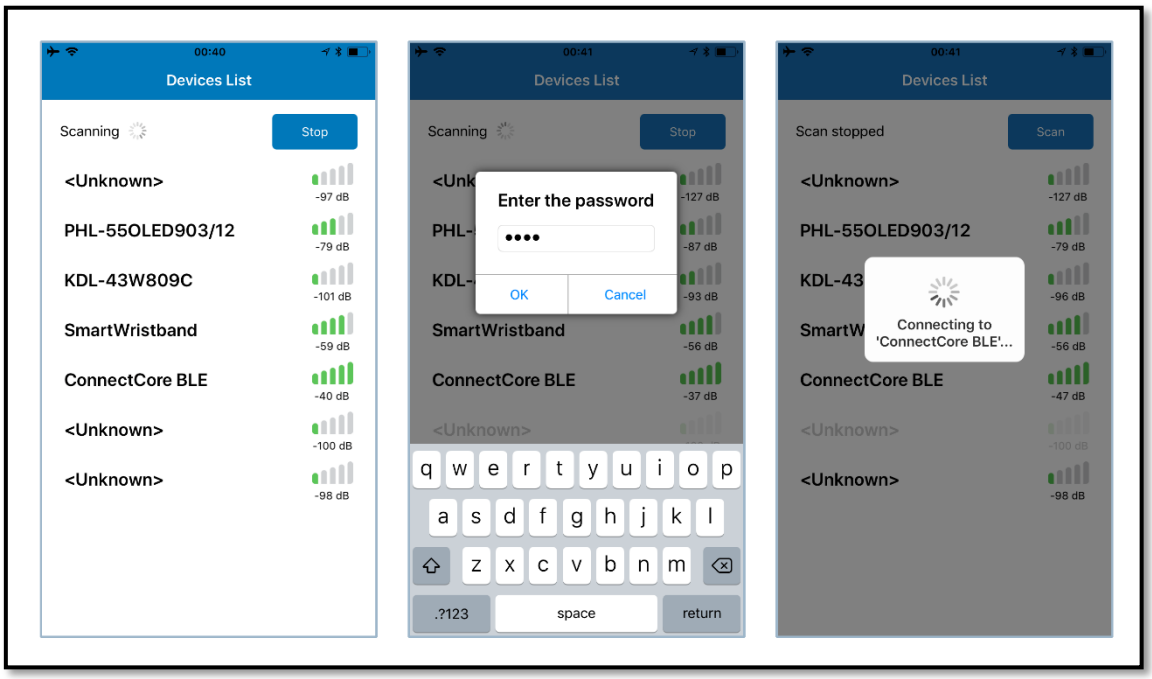

Figura 20. Aspecto final de la página de inicio de la aplicación móvil

La página de inicio, en la que se lleva a cabo el rastreo de dispositivos, está compuesta por:

- La vista **DeviceListPage.** Sus métodos, atributos y propiedades de interés son:
  - En el XAML, una **lista** de tipo grid, cuyos elementos se van actualizando según se descubren nuevos dispositivos BLE por medio de enlaces con el modelo de vista.
  - El método **OnItemSelected**, que se ejecuta cuando se selecciona uno de los dispositivos de la lista. Prepara al modelo de vista para que comience la comunicación con dicho dispositivo.

- El modelo de vista **DeviceListPageViewModel**. Sus métodos, atributos y propiedades de interés son:
    - La variable **adapter** de tipo IAdapter, de la librería Plugin.BLE. Representa el adaptador BLE, por lo que permite gestionar todo lo relacionado con esta interfaz, como el descubrimiento de nuevos dispositivos, o comprobar si se ha perdido la conexión con un dispositivo.
    - La propiedad **Devices**, una lista en la que se van almacenando todos los dispositivos que son descubiertos por la aplicación, con el método **AddDeviceToList**. Cada vez que la lista se actualiza, se informa a la vista con el evento OnPropertyChanged. Una vez se selecciona un dispositivo se queda guardado en **SelectedDevice**.
    - El método **InitPage**, al que se llama al crear la página por primera vez, suscribe a adapter a dos eventos: el evento de descubrimiento de un nuevo dispositivo, de forma que el método manejador lo añada a Devices, y el evento de conexión perdida, de forma que el método manejador cierre la conexión con SelectedDevice y haga que la aplicación vuelva a la página de descubrimiento de dispositivos si no estaba en ella.
    - El método **ConnectToDevice** abre la conexión con SelectedDevice y gestiona si todo ha ido bien durante este proceso y durante la autenticación. Mientras tanto también muestra un diálogo de carga, evitando así que el usuario ejecute cualquier otra acción. Finalmente, una vez que la comunicación se abre correctamente, avanza hasta la siguiente página, de tipo DevicePage. Si en su lugar hubiera habido cualquier error, lo mostraría dentro de un pop-up.
    - Antes de la autenticación, el método **AskForPassword** crea una nueva página de tipo PasswordPage a modo de pop-up, en el que permite al usuario introducir una contraseña y autenticarse contra SelectedDevice.

En la Figura 21 se muestra el aspecto final de la pantalla de información del dispositivo.

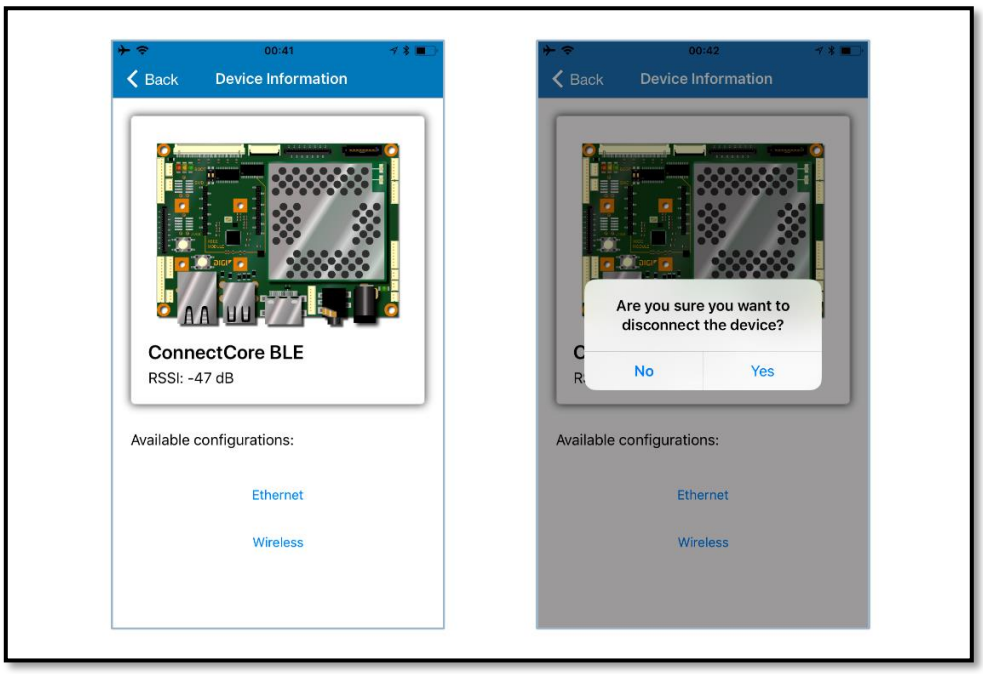

Figura 21. Aspecto final de la página de información del dispositivo de la aplicación móvil

La página en la que se muestran las interfaces de red y demás información del dispositivo seleccionado, está compuesta por:

- La vista DevicePage. Sus métodos, atributos y propiedades de interés son:
  - La variable **device** guarda el objeto de tipo BLEDevice asociado a la página.
  - En el constructor de la clase se sobrescribe el botón de regreso a la página anterior para que al pulsarlo muestre un pop-up en el que pregunte al usuario si desea desconectarse del dispositivo.
- El modelo de vista DevicePageViewModel. Sus métodos, atributos y propiedades de interés son:
  - La variable **device** almacena el mismo BLEDevice que la vista.
  - Las propiedades **MacAddress**, **Name** y **Rssi** enlazadas con la vista.
  - El método **DisconnectDevice**, que desconecta el dispositivo. Es llamado en el pop-up del botón de retorno.
  - El método **OpenNetworkInterface**, en el que se avanza a la página de configuración de la interfaz de red. Este es el método al que se llama cuando se pulsa uno de los botones, y según cuál sea se le pasa un valor u otro ("Ethernet" o "WiFi").

En la Figura 22 se muestra el aspecto final de la pantalla de configuración de la interfaz de red.

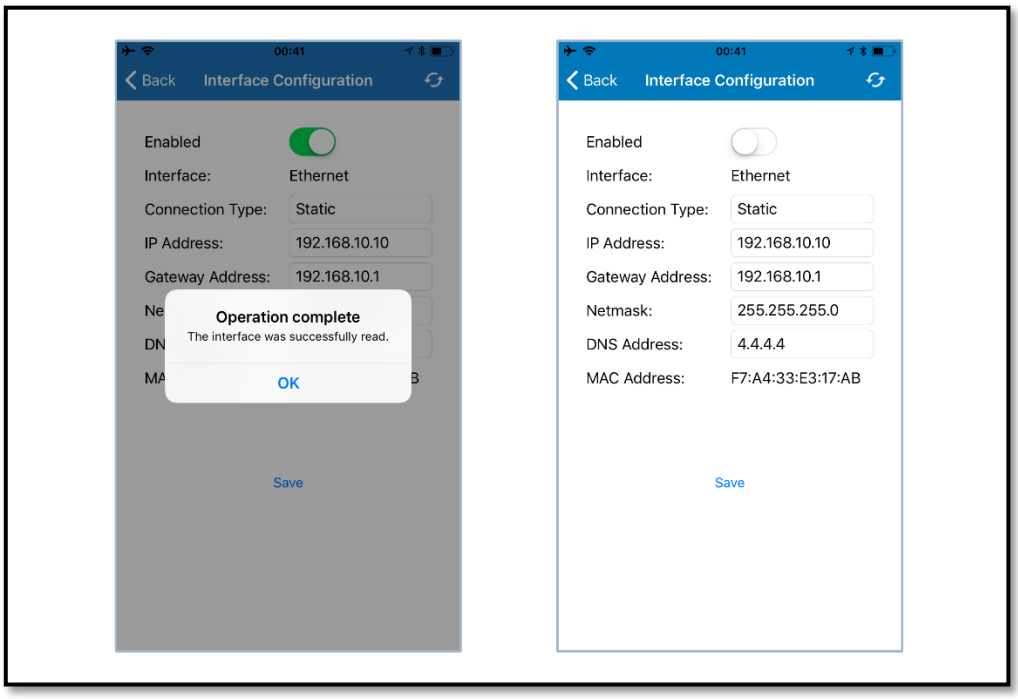

Figura 22. Aspecto final de la página de configuración de una interfaz de la aplicación móvil

La página en la que se muestran y se permite cambiar las propiedades de la interfaz de red seleccionada, está compuesta por:

- La vista InterfacePage. Sus métodos, atributos y propiedades de interés son:
    - La propiedad Enabled de los campos de texto enlazados a IpAddress, DefaultGateway y SubnetMask está enlazada al valor de ConnectionType; cuando su valor es “Dynamic” se impide que el usuario introduzca nuevos valores en esos campos. Para conseguir este comportamiento se crea la clase **ConnectionTypeConverter**, que hereda de la interfaz IValueConverter.
    - También en el XAML hay presentes dos botones, uno para enviar los nuevos valores de las propiedades al dispositivo conectado y otro para recargar y volver a leer los valores. Ambos utilizan enlaces con Commands.
    - La variable **device** almacena el mismo BLEDevice que DevicePage y DevicePageViewModel.
    - El método **OnAppearing**, que se ejecuta cuando aparece por primera vez la página. En él se llama al método OpenConnection de InterfacePageViewModel.
- El modelo de vista InterfacePageViewModel. Sus métodos, atributos y propiedades de interés son:
    - La variable **device** almacena el mismo BLEDevice que la vista.
    - Los métodos **OpenConnection** y **CloseConnection**, que suscriben y dan de baja del evento UserDataRelayReceived del XBeeDevice de BLEDevice, que se dispara cada vez que se recibe un nuevo dato a teavés de la interfaz BLE.
    - El método **ReadValues**, que crea una cadena JSON con la petición de lectura. La cadena se envía a través de la interfaz BLE con SendSerialData del XBeeDevice. Finalmente, el método se bloquea mientras espera una señal de confirmación para comprobar si los datos se han recibido correctamente, y muestra mientras tanto un diálogo de carga. Si la operación ha sido exitosa lo notifica al usuario, y si no lo ha sido, o se ha acabado el tiempo de espera, le muestra un pop-up con los mensajes de error encontrados.
    - El método **WriteValues**, que crea una cadena JSON que contiene una petición de escritura con todos los valores contenidos en las propiedades enlazadas. Comprueba si los datos del JSON producido se adecuan al formato previamente establecido mediante el método **CheckJson**. Si son incorrectos muestra un pop-up con todos los errores encontrados, y si son correctos, envía el mensaje. Finalmente, espera hasta recibir una señal de confirmación.
    - El método **UserDataRelayReceived**, que sirve como manejador del evento UserDataRelayReceived del XBeeDevice de BLEDevice. Este método gestiona todos los mensajes recibidos después de enviar mensajes en los dos métodos anteriores. Si la operación era de lectura, y ha sido exitosa, actualiza los valores de las propiedades enlazadas y envía una señal de confirmación a ReadValues. Si la operación era de escritura, envía una señal a WriteValues.

Una vez completada esta implementación, se encontró un problema importante al desplegarlo en el iPhone: el tamaño máximo admitido en mensajes de Bluetooth Low Energy es de 185 bytes en iOS 10, la versión de iOS utilizada, a pesar de que en versiones posteriores no exista esta limitación. Para solucionar este problema, ha sido necesario modificar ligeramente el formato del JSON utilizado hasta ahora y cambiarlo por el siguiente:

```
{
  "Operation" : "Read" / "Write",
  "Interface" : "Ethernet" / "WiFi",
  "Type" : "Static" / "Dynamic",
  "IP" : "xxx.xxx.xxx.xxx",
  "Gateway" : "xxx.xxx.xxx.xxx",
  "Netmask" : "xxx.xxx.xxx.xxx",
  "DNS" : "xxx.xxx.xxx.xxx",
  "MAC" : "xxx.xxx.xxx.xxx",
  "Enabled" : "True" / "False",
  "Status" : "Error" / "OK"
  }
```

Figura 23. Estructura del protocolo JSON

El mayor cambio es que se prescinde de un segundo servidor DNS, aunque no es muy importante dado que se trata de un ejemplo para el uso del API y no de una aplicación para configurar interfaces de red reales. El resto son solo pequeños cambios en los nombres de los campos.

Además de esto, para conseguir una reducción suficiente en el tamaño de los datos, se ha optado por comprimir cada cadena antes de enviarla y descomprimirla al recibirla. El formato de compresión utilizado es Deflate. Se generan datos deflate raw, sin encapsular, para obtener la máxima relación de compresión posible a costa de perder cabeceras para la identificación y detección de errores. Esto no supone un problema ya que la validez de los datos se comprueba a cada lado de la comunicación igualmente. Tanto Python como C# ofrecen librerías para trabajar con este formato:

- En Python se utilizan las funciones compress y decompress de la librería zlib. Esta librería añade dos bytes de cabecera al comprimir, por lo que es necesario extraerlos del vector de bytes producido antes de enviarlo.
- En C#, se crea la clase Compression con dos métodos estáticos, Compress y Decompress. Estos utilizan la clase DeflateStream de System.IO.Compression para comprimir y descomprimir secuencias de bytes.

2.6.2 CCBLESDK-25 – Prueba de componentes de la aplicación móvil

Una vez acabado el resto de tareas de desarrollo, lo que queda es probar todo lo implementado hasta ahora. Esto supone ejecutar las pruebas de componentes del ejemplo completo, incluyendo la aplicación Python embebida y la aplicación móvil que se conecta a ella.

En las Tablas 28 y 29 se recuperan las pruebas definidas en la Tabla 20 del apartado 2.4.5 Y en la Tabla 25 del apartado 2.5.5 respectivamente, junto con los resultados de las pruebas.

Código	Prueba	Interfaz nativa	Interfaz XBee
PP-01	Comprobar que la aplicación se conecta correctamente con un cliente.	Superada	Superada
PP-02	Comprobar que la aplicación muestra correctamente la interfaz Bluetooth utilizada.	Superada	Superada
PP-03	Comprobar que la aplicación obtiene correctamente los datos de los falsos ficheros de configuración y que el cliente los recibe correctamente tras una petición.	Superada	Superada
PP-04	Comprobar que la aplicación modifica correctamente los datos de los falsos ficheros de configuración cuando el cliente hace una petición, y que este recibe un mensaje de confirmación.	Superada	Superada
PP-05	Comprobar que, tras cambiar el nombre anunciado, este se muestra correctamente.	Superada	No superada

Tabla 28. Resultados de las pruebas de la aplicación Python

Código	Prueba	Interfaz nativa	Interfaz XBee
PA-01	Comprobar que la aplicación rastrea correctamente dispositivos BLE anunciándose.	Superada	Superada
PA-02	Comprobar que la aplicación se conecta correctamente con un servidor.	Superada	Superada
PA-03	Comprobar que la aplicación detecta correctamente las interfaces de red que permite configurar el servicio.	Superada	Superada
PA-04	Comprobar que la aplicación lee los datos de una interfaz implementada en el servidor.	Superada	Superada
PA-05	Comprobar que los datos se actualizan correctamente tras modificar la configuración de una interfaz.	Superada	Superada

Tabla 29. Resultados de las pruebas de la aplicación móvil

Durante la ejecución de las pruebas se han encontrado algunos errores.

El primero es el relativo a la prueba PP-05, que viene arrastrado desde las pruebas del API. Como la versión del firmware del XBee es antigua, no es posible comprobar que funcione bien el cambio de nombre.

El segundo es un problema relativo al API que no se detectó hasta que se han realizado pruebas más exhaustivas. Al fallar la autenticación en una comunicación a través de la interfaz nativa se ha observado que el API tardaba demasiado tiempo en responder informando del fallo. La causa se ha encontrado en un pequeño fallo en la parte del código encargada de gestionar el error de autenticación, y una vez resuelto la notificación es mucho más rápida.

2.6.3 Revisión del sprint

La información sobre las horas dedicadas a cada tarea se recoge en la Tabla 30.

Tarea	Horas estimadas	Horas reales	Desviación
Implementar aplicación móvil	20	34	+70 %
Prueba de componentes de la aplicación móvil	12	4	-66,7 %
Documentación	8	8	0%

Tabla 30. Comparativa de horas dedicadas en el sprint 6

Durante este sprint no ha sido posible acceder a la cuenta de la empresa, ni tampoco a la página de Jira del proyecto. Por esta razón, no se han podido marcar como completadas las tareas en el tablero Scrum de Jira y los gráficos obtenidos no corresponden con el trabajo real.

La implementación de la aplicación móvil ha llevado aproximadamente dos semanas por ser una tarea bastante extensa y por los problemas encontrados, como se explica en el apartado 2.5.1.

Como se previó que esto podía pasar, se utilizaron 4 días adicionales en el sprint, que son los que se han dedicado a las pruebas del sistema. Esta tarea transcurrió con normalidad, e incluso se cerró con algunos días de margen.

Aunque se comentó que el sprint mantendría la duración inicial de 40 horas, se ha dedicado algo más de tiempo para acabar el desarrollo definitivamente.

Y con todo esto llega el fin del último sprint. Todas las historias y tareas definidas para el proyecto han sido completadas antes de la fecha límite.

Capítulo 3. Conclusiones

Evaluación del proyecto

La Tabla 31 reúne las comparativas entre los tiempos estimados para cada tarea y el tiempo dedicado realmente, según ha sido recogido durante el desarrollo. Cada tiempo incluye el dedicado tanto al desarrollo como a la documentación.

TAREA		TIEMPO ESTIMADO	TIEMPO REAL	DESVIACIÓN
Gestión		**25 h**	**26 h**	**+4 %**
	Planificación	20 h	20 h	0 %
	Seguimiento	5 h	6 h	+20 %
API Python		**120 h**	**150 h**	**+25 %**
TA-01	Análisis del API	5 h	14 h	+180 %
TA-02	Diseño del API	15 h	16 h	+6,7 %
TA-03	Creación de servidor GATT	15 h	10 h	-33,3 %
TA-04	Conexión BLE	30 h	32 h	+6,7 %
TA-05	Protocolo de comunicación	15 h	14 h	-6,7 %
TA-06	Capa de seguridad	25 h	56 h	+124 %
TA-07	Pruebas del API	15 h	8 h	-46,7 %
Aplicación Python		**60 h**	**42 h**	**-30 %**
TP-01	Análisis de la aplicación	5 h	2 h	-60 %
TP-02	Diseño de la aplicación	10 h	4 h	-60 %
TP-03	Interfaz BLE	20 h	30 h	+50 %
TP-04	Fichero de propiedades	15 h	4 h	-73,3 %
TA-05	Pruebas de la aplicación	10 h	2 h	-80 %
Aplicación móvil		**60 h**	**52 h**	**-13,3 %**
TM-01	Análisis de la aplicación móvil	5 h	2 h	-60 %
TM-02	Diseño de la aplicación móvil	10 h	6 h	-40 %
TM-03	Interfaz BLE	10 h	30 h	+200 %
TM-04	Interfaz gráfica	25 h	12 h	-52 %
TM-05	Pruebas de la aplicación móvil	10 h	2 h	-80 %
Memoria (Introducción y Conclusiones)		**35 h**	**55 h**	**+57,1 %**
TOTAL		**300 h**	**325 h**	**+8,3 %**

Tabla 31. Comparativa de tiempos del proyecto

Hay dos casos especialmente destacables, en los que el porcentaje de desviación ha sido de más del 100%. Uno de ellos se atribuye a una mala estimación: para TA-01 el análisis del API no fue tan trivial como se esperaba. El otro caso corresponde con la implementación de la capa de seguridad del API y supone la mayor desviación con diferencia. Debido a causas externas al proyecto (la falta de acceso al material necesario durante el confinamiento por COVID-19), se invirtieron unas 30 horas más de las esperadas en esta tarea ya que no era posible continuar con nuevas tareas de implementación hasta acabarla.

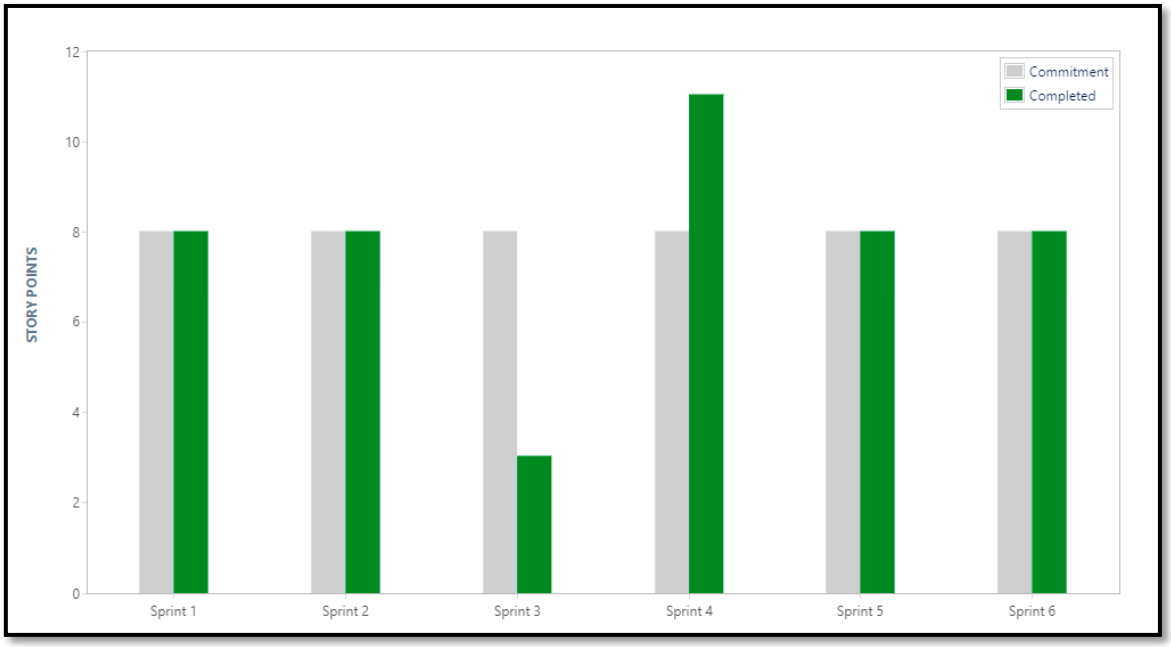

Figura 24. Puntos de historia completados en el proyecto

En el diagrama de la Figura 24 aparece un resumen del rendimiento durante el proyecto, esta vez considerando los puntos de historia asignados a cada sprint frente a los completados. Pese a haber sido un desarrollo algo accidentado, se han cumplido todos los plazos excepto en el sprint 3, cuando empieza la implementación de la capa de seguridad.

Por lo general, se puede decir que el proyecto se ha completado con éxito. Las tareas adicionales con menos prioridad que están expuestas en la Tabla 4 no se han podido abordar, pero todo aquello que era imprescindible para el SDK está hecho. Resulta especialmente bueno teniendo en cuenta la situación excepcional en la que se ha desenvuelto más de la mitad del desarrollo, que ha forzado al teletrabajo y ha limitado la capacidad para afrontar algunos de los problemas que en la oficina se habrían resuelto mucho más fácilmente. Como además la empresa pretende utilizar este SDK en futuros proyectos, es satisfactorio haber podido entregar un proyecto totalmente funcional.

Lecciones aprendidas

Este proyecto ha abarcado el desarrollo de un SDK completo, incluyendo tanto un API destinado a ser usado en una computadora ConnectCore, como un ejemplo de aplicación móvil multiplataforma que permita la conexión a un servidor que haga uso de dicho API. El hecho de que haya sido un proyecto tan amplio me ha permitido introducirme a muchas nuevas tecnologías, y afianzar conocimientos de otras que ya había estudiado durante la carrera.

Desde la parte del desarrollo del API, he podido profundizar en el lenguaje Python, en su sintaxis y su filosofía de diseño; he entrado en contacto con paquetes de datos y protocolos de comunicación en un entorno real; he aprendido cómo funciona Bluetooth Low Energy, cuáles son sus limitaciones y cómo trabajar con él; he aplicado protocolos de seguridad y cifrado de datos; y he estado trabajando con sistemas embebidos, comunicándome por consola de comandos con un ConnectCore y a través de este con un XBee.

Por la parte de la aplicación móvil, puse en práctica lo aprendido sobre C#, aprendí sobre programación asincrónica en este lenguaje, y utilicé el patrón MVVM en el diseño.

Además, a lo largo de todo el desarrollo he estado sincronizando el proyecto en un repositorio Git, lo cual me ha permitido comprender un poco mejor el funcionamiento de los sistemas de control de versiones, al mismo tiempo que he podido aprender a reaccionar frente a casos menos frecuentes.

Por otra parte, este trabajo también me ha servido para pulir otras competencias transversales de gran importancia.

Para la gestión de futuros proyectos, he aprendido a utilizar la herramienta Jira y a obtener datos útiles sobre el rendimiento, he podido practicar estimaciones en un ámbito real, en el que todos los miembros del equipo daban su opinión acerca del número de puntos de historia que cada tarea debía tener, mientras defendían su posición, y he comprendido un poco mejor la importancia de una buena planificación, teniendo en cuenta siempre los posibles contratiempos que se van a encontrar.

También se ha visto un refuerzo del trabajo tanto en grupo como individual. A lo largo del proyecto, la puesta en común de los problemas que iba encontrando ha sido imprescindible para llegar a una buena solución entre todos los miembros del equipo, hasta tal punto que el desarrollo se habría detenido en algunos casos excepcionales si no hubiera pedido ayuda. Aun así, la mayoría de los contratiempos se resolvieron de forma autónoma, lo que supuso en muchas ocasiones explorar nuevas tecnologías por mi cuenta, como fueron todo lo relacionado con el D-Bus o formatos de compresión.

Futuras mejoras

Aunque se cumplieron los objetivos principales del API, aún hay otros aspectos del producto que podrían mejorarse. Algunos de estos aspectos provienen del hecho de que no fue posible probar correctamente ciertas funciones a causa del teletrabajo y la falta de recursos:

- La aplicación móvil fue desarrollada en Xamarin.Forms, y como tal es una aplicación multiplataforma. Sin embargo, debido a la extraña limitación que apareció en el MTU de la comunicación BLE al usar dispositivos Android (que hizo imposible enviar datos de más de 20 bytes), y al no poder hacer pruebas con más dispositivos diferentes, se limitó el desarrollo a iOS. La lógica de negocio debería funcionar correctamente, pero faltan por implementar ciertos aspectos personalizados para la interfaz gráfica de Android, y es necesario hacer pruebas para determinar si realmente es posible resolver el problema.
- La versión del firmware del XBee utilizado en pruebas no incluía el parámetro para cambiar el nombre anunciado por BLE, por lo que convendría probarlo con versiones más actualizadas.

Además, por otra parte, quedan pendientes algunas otras funciones extra que aumentarían el valor del producto o facilitarían futuros desarrollos similares en la empresa, como por ejemplo:

- Crear una nueva librería extrayendo clases y métodos de la librería XBee que hayan sido utilizadas en el desarrollo.
- Permitir al usuario cambiar entre la interfaz utilizada en la comunicación (nativa o XBee) a voluntad en tiempo de ejecución.
- Relacionado con lo anterior, actualmente si un ConnectCore tiene soporte Bluetooth y un XBee conectado, se utiliza la interfaz nativa por defecto. Sería conveniente permitir al usuario elegir cuál desea usar desde el principio.

Bibliografía

Adafruit Industries. (20 de Marzo de 2014). *Introduction to Bluetooth Low Energy*. Obtenido de https://learn.adafruit.com/introduction-to-bluetooth-low-energy

Byford, B. (s.f.). *python-bluezero.* Obtenido de https://github.com/ukBaz/python-bluezero

Digi Internacional Inc. (s.f.). *XBee Python Library*. Obtenido de https://xbplib.readthedocs.io/en/latest/index.html

Digi International Inc. (s.f.). *BLE Unlock API Specification*. Obtenido de https://www.digi.com/resources/documentation/Digidocs/90002258/reference/r_frame_0x2c.htm

Digi International Inc. (s.f.). *Documentación de Digi*. Obtenido de https://www.digi.com/support/supporttype?type=documentation

Digi International Inc. (s.f.). *Módulos y accesorios Digi XBee 3*. Obtenido de https://www.digi.com/products/embedded-systems/digi-xbee/rf-modules/2-4-ghz-modules/xbee3-zigbee-3#partnumbers

Microsoft Corporation. (s.f.). *Patrón Model-View-ViewModel*. Obtenido de https://docs.microsoft.com/es-es/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

Pennington, H., Carlsson, A., Larsson, A., Herzberg, S., McVittie, S., & Zeuthen, D. (s.f.). *D-Bus Specification*. Obtenido de https://dbus.freedesktop.org/doc/dbus-specification.html

Python Software Foundation. (s.f.). *28.8. abc — Abstract Base Classes*. Obtenido de https://docs.python.org/2/library/abc.html

Universidad Stanford. (s.f.). *What is SRP?* Obtenido de http://srp.stanford.edu/whatisit.html